CHAPTER 9

# Functional verification

Hung-Pin (Charles) Wen
*National Chiao-Tung University, Taiwan*

Li-C. Wang
*University of California, Santa Barbara, California*

Kwang-Ting (Tim) Cheng
*University of California, Santa Barbara, California*

## ABOUT THIS CHAPTER

In a typical **integrated circuit** (IC) design flow, functional verification ensures that the implementation conforms to the specification. Because of the rapid growth of both design size and complexity, functional verification has become one of the key bottlenecks in the design process. For example, it has been reported in [Bailey 2002] that the functional verification process consumes more than 70% of the design effort, and this number might continue to increase. Functional verification is critical, because an undetected bug in a design may result in significant financial loss for a company. The Pentium recall for the famous FDIV bug, for example, cost Intel more than $450 million in 1995. Therefore, effective verification strategies and techniques have become indispensable to the design flow to ensure high verification quality.

This chapter starts with an overview of the basic concepts of functional verification and its general flow. Current challenges are explained to help readers to understand the complexity of functional verification. Meanwhile, modern designs usually follow the principle of hierarchism by decomposing a complex system into multiple components. Each decomposition boundary is referred to as a **level**. A brief discussion of verification at each of these levels is introduced.

To assess the verification quality, coverage metrics are developed for measuring the extent of an intended verification task. Coverage metrics can be divided into two categories: structural and functional. Structural coverage metrics calculate a coverage number on the basis of specific structural representations, such as lines and branches, in the hardware description model and are the most popular measures. Functional metrics, on the other hand, focus on the semantics or the design intent of the hardware description model. In this chapter, various structural coverage metrics will be reviewed in detail.

**Simulation-based verification** is the most widely used approach in functional verification. Simulation is based on testbenches. In a typical verification task, testbenches accompanied with a design description model are developed and include input stimuli and expected output responses by the design. The efficiency of the simulation determines the efficiency of the verification, and, hence, having compact and high-quality stimuli is critical to this approach. An alternative to simulation-based verification is **formal verification**. Formal verification relies on mathematical reasoning techniques to verify a design. There can be two types of formal verification methods, one to prove specific properties of a design and the other to prove that two models of a design are equivalent. The former is called property checking, and the later is often referred to as equivalence checking. At the end of this chapter, some of these formal verification techniques will be introduced as supplemental materials.

## 9.1 INTRODUCTION

Verification processes happen everywhere in our daily life. One general definition of verification given in [ANSI/ASQC 1978] is "the act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services or documents conform to specified requirements." Within the context of design automation of IC design, shown in Figure 9.1, functional verification is the step to ensure that the specifications

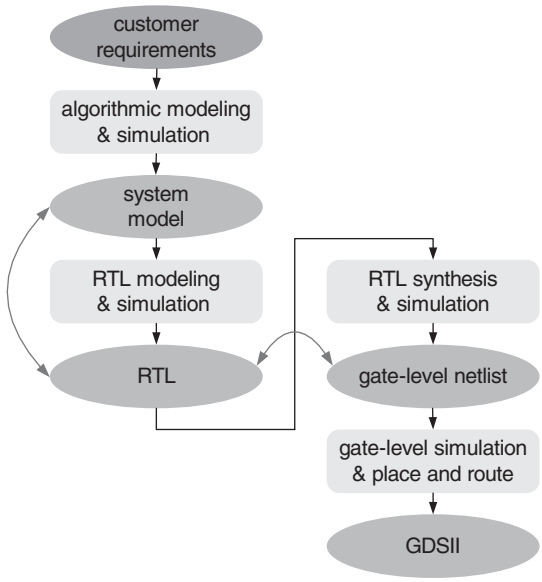



**FIGURE 9.1**

Typical design flow overview.

and/or the implementations of the design at various abstraction levels are in accord with the design intent.

In a typical design flow, representations for a design at different abstraction levels often contain thousands of lines or more of ***Hardware Description Language*** (HDL) code. These representations are error-prone because of the high complexity of the design. Verification plays an important role in identifying various kinds of problems that may have occurred at different design stages. For many medium-scale to large-scale processors, ***application-specific integrated circuits*** (ASICs), or ***system-on-chips*** (SOCs), functional verification can consume more than 70% of the total labor effort in the design process [Piziali 2006]. The difficulty inherent in functional verification is a result of the following three issues:

1. **Ambiguous specifications:** Customer requirements are often written colloquially into the specification. It may be difficult to precisely specify the requirements with a natural language such as English. Moreover, a specification is often described at the system level. When verifying a unit or block inside a system, a clear specification for the unit or the block usually is not available.
2. **Complexity explosion:** In general, the complexity of a Boolean circuit can grow exponentially in terms of both the number of inputs and the number of internal states. Exhaustive simulation (of all input value combinations and/or state combinations) is simply infeasible for any nontrivial design.
3. **Quality concerns**: Ensuring highest-quality verification with limited engineering resources and within limited time is the challenge to every verification task. To effectively use resources and time, one needs coverage metrics to guide the spending of verification effort. Although various coverage metrics exist to measure verification coverage, none of these metrics have been shown to be the golden metric that can reliably and accurately reflect the verification quality. As a result, signing off a design with respect to functional verification can become a managerial decision that heavily depends on one's experience and is often influenced by time-to-market pressure as well.

## 9.2 VERIFICATION HIERARCHY

Modern IC designs typically follow a **top-down** implementation flow in which a system is hierarchically partitioned into components. Each partitioning boundary defines the level of the design components. Within the hierarchy, verification tasks need to be performed before individual components are assembled. The V diagram in Figure 9.2 illustrates the design, verification, and integration

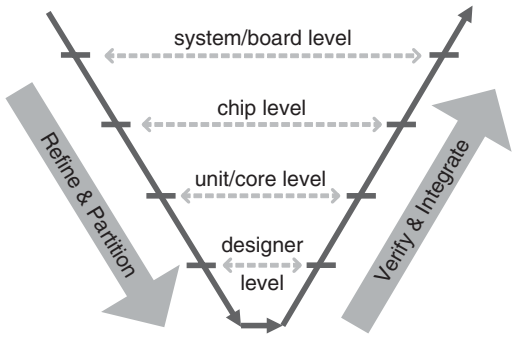



**FIGURE 9.2**

V diagram of design, verification, and integration.

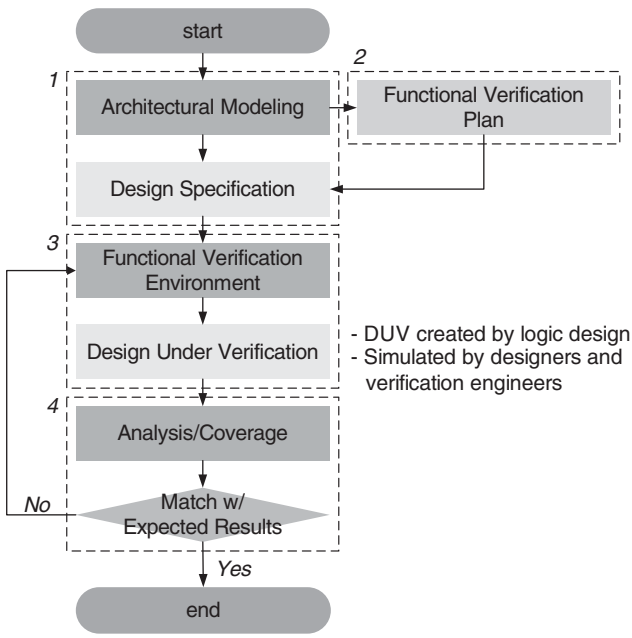



**FIGURE 9.3**

Generic design verification flow.

flow starting from the system/board level, through the chip and core/unit levels, to the designer level.

A generic verification flow [Palnitkar 2003a] for each level consists of several steps, as shown in Figure 9.3. In Step 1, architects need to prepare a design specification for the best architecture on the basis of analysis of simulation

result. In Step 2, a functional verification plan is created to define the basic parameters that are used later in the functional verification environment. Test vectors and testbenches are either generated manually or automatically by tools during Step 3. A software simulator applies these test vectors and testbenches to the ***design under verification*** (DUV) and collects the related information after simulation. In Step 4, the output data are analyzed and checked against the expected results to calculate verification coverage. If the desired coverage goal is not achieved, Step 3 is repeated to generate more test vectors to improve the coverage. After the coverage goal is met, optional steps of hardware-accelerated simulation, emulation, and assertion-based verification could be applied to further improve verification quality and to reduce the risk of needing a future re-spin.

### 9.2.1 Designer-level verification

In the top-down implementation flow shown in Figure 9.2, the designer level is the lowest level that defines the smallest of the RTL modules such as an arbiter or a **first-in first-out** (FIFO) that one designer can be in charge of in a project. Designer-level blocks are usually verified individually to ensure that the basic functionalities of the blocks understood by the designer from the system specification are correctly implemented. As the tasks involved in verifying a designer-level block do not require interaction with other blocks, the designer is given full control of the block, and thus a high standard of verification is expected at this level.

During the early phase of a design project, the functionality of a block would not be completely fixed and likely will be modified frequently. For example, part of a block's functionality may need to move across the interface to other blocks for better unit/core/chip optimization. It is, therefore, not uncommon to repeat the designer-level verification process multiple times.

A variety of verification techniques are available at this level. Testbench development is relatively easy because the block inputs and outputs are treated as primary inputs and outputs at this stage. The designers often explore most of or even the entire input space of the target block by simulation. Formal methods such as property checking can also be applied relatively easily at this level because of the small design size. It is important to note that, for designer-level verification, the main challenge is not in verifying the block itself as an independent design, but in verifying the block in the context of the environment in which it will be placed. For example, a property may not be verified as always true if the block operates independently. However, under specific constraints imposed by the environment surrounding the block, the property could become always true. Establishing proper environmental constraints for designer-level verification is, therefore, an important (and usually not trivial) task.

### 9.2.2 Unit-level verification

A complex design is usually divided into several logical components that are referred to as **units**. The units intercommunicate through buses following pre-specified protocols. Figure 9.4 shows an AMBA bus-based SOC design. Memory, UART, Bridge, and Arbiter are among the units created from many different designer-level components. In this example, the communications between units go through two PCI buses. [Scafidi 2004] reported that even when the full-chip model of Intel's Itanium-2 processor was close to the tape-out quality, unit-level verification still uncovered additional bugs.

The functionality at the unit level is specified more clearly, and usually the specification is more stable than that at the designer level. Each unit usually has a precise specification where its physical and timing characteristics will abide by the requirements of the bus protocol. Each unit implements a set of specified operations. Therefore, the goal of unit-level verification is to guarantee that each operation performed by the unit conforms to the desired functionality and satisfies the bus interface's communication constraints.

Because of the high accessibility of units through buses, high-quality verification that guarantees each unit correctly meets its formal specification is usually achievable. In an ideal situation, once the unit-level verification is completed, bugs residing within these units can be excluded from the list of candidates. When performing verification at the next level, only those bugs originated from the communication and physical interfaces need to be considered.

### 9.2.3 Core-level verification

In the example of Figure 9.4, units such as the ARM processor core, the DMA core, and the third-party IPs are initially designed for general purpose use and are equipped with more generalized functionalities. They are incorporated into

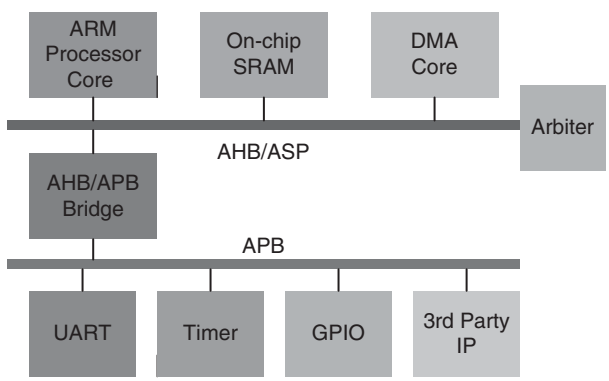



**FIGURE 9.4**

AMBA bus-based SOC.

an SOC design to avoid the need for developing dedicated logic, which often requires only a subset of the original functionalities. Such reusable components are referred to as **cores** and can be either acquired from other companies or developed internally in a company. In modern SOC designs, a core is often used multiple times within a system or across different systems. For core providers, it is necessary to thoroughly verify the functionality of the core before it is delivered to the core integrators.

Cores are often designed as a stand-alone component in the first place. In addition to core-specific functionalities, standardized bus protocols and/or physical interface standards are then incorporated to offer core reusability. The corresponding verification components used to stimulate and monitor these standard buses or interfaces can, therefore, be reused and shared among cores by use of the same bus protocols or physical interfaces.

Even if a core has its own stand-alone specification, this specification can change because of bug fixing or functionality enhancement, either of which may alter the original functionality. Therefore, it is necessary to re-ensure that operations defined in a previous version of the core will still work correctly in a subsequent version. This requirement is called **backward compatibility**. To meet this requirement, a **regression test suite** is commonly used. Such a test suite is developed by collecting interesting and useful tests from verification conducted on previous versions of the design. A new version must pass these tests to ensure backward compatibility. Note that if a bug exists in old versions of the design, we should not expect regression tests to capture this bug in the new version even if a fix to the bug has been inserted. For that purpose, new tests are required to verify the correctness of the inserted fix.

### 9.2.4 Chip-level verification

A chip-level design consists of multiple units/cores that have complete RTL and bus functional models with well-defined I/O boundaries. At this level, the specification usually does not change significantly from its initial architecture. Hence, the verification requirement is usually well defined.

The aim of chip-level verification is to ensure that the components are properly connected through the interfaces and the entire design abides by the specification. For a regular interface structure such as a bus protocol, only a restricted set of sequences of control and data signals, typically called **transactions**, are permitted. On the basis of the specified interactions between the units, transaction-based tests can be developed to verify the interfaces.

A transaction-based test usually consists of one top-level RTL file that includes all units and bus interfaces and one testbench file that produces transactions to propagate events from one unit to another through the bus interface. Responses at the primary I/Os and/or memory contents are monitored to check the overall behavior of the system.

### 9.2.5 System-/board-level verification

System-level integration is a complex task that requires many tools for design creation, simulation, and analysis. In [Bailey 2007], system-level verification is defined as "the utilization of appropriate abstractions to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner."

Verification at this level involves checking the integration through the interconnections between different chips on the board. The functionality at the lower levels is assumed to have been fully verified. Often, the application software is applied at this stage to verify the entire system.

Verification engineers frequently use programmable logic devices, such as ***field programmable gate arrays*** (FPGAs), to emulate the design. With the design implemented in programmable devices, the testbenches can be executed directly on such emulated implementations, which is significantly faster than executing the testbenches with a software simulator.

## 9.3 MEASURING VERIFICATION QUALITY

"When can one claim that the verification is complete?" This is a perpetual and still unanswerable question. Even if a verification team performs all the scheduled tasks, and even if no more new bugs can be discovered over an extended verification period, say a few weeks, there is no guarantee that additional simulation would not discover a new bug. The total space to be verified is well beyond what can be exhaustively simulated. Considering a logic block with 64-bit inputs, the combinatorial possibilities for its input space reach $16 \times 10^{18}$ billion. If simulating one instance takes one nanosecond, then simulating all of them will take 5.07 centuries. Obviously, some modeling, analysis, and optimization techniques need to be used to avoid simulating all tests exhaustively. Various measures are developed to guide the selection of tests for simulation. These measures are typically referred to as **coverage metrics**. Rather than simulating all tests, the idea is to simulate just enough tests to reach a desired coverage goal on the basis of the given metric. The assumption is that achieving the coverage goal implies that a sufficient verification quality has been accomplished.

In this section, we will first introduce the concept of random testing followed by the coverage-driven verification paradigm to outline the concept of coverage in verification. We will also introduce a classification of verification metrics and common coverage metrics within each category.

### 9.3.1 Random testing

Random testing is the most intuitive verification approach. A test generation program is used to generate random tests according to a set of test templates

along with a seed. Multiple random instances of each test template are generated and applied to exercise a variety of scenarios for exploring various design corners. A refinement of this approach, called **constrained random verification**, relies on a collection of additional constraints to guide the generation of tests. Figure 9.5 illustrates the concept of the random testing approach.

Random test generation requires two types of inputs to **constrain** the test generation process: (1) a template that serves as the skeleton of the test case, which contains a set of unknown input fields, and (2) a set of arguments for which the values can be set during the generation process. Instead of handcrafting tests directly, users specify these arguments for input fields within their legal ranges. Multiple instances of physical test cases are then automatically generated from each template by specifying values in the input fields. Templates, along with the changeable arguments, provide an abstract mechanism for hiding the structural details from users while simultaneously satisfying all architectural constraints.

Take microprocessor verification as an example. Its test template is an assembly program with a set of predefined bias arguments. On the basis of these parameters, one can create arguments to:

1. Select an instruction,
2. Select the next instruction on the basis of the current one,
3. Select an operand,
4. Use branch and jump,
5. Cause an overflow or underflow,
6. Interrupt to cause an exception.

However, all the preceding arguments must conform to the architectural constraints, such as, for instance, 32 registers (20 general-purpose, 12 special-purpose), 24-bit addressing, and indirect addressing.

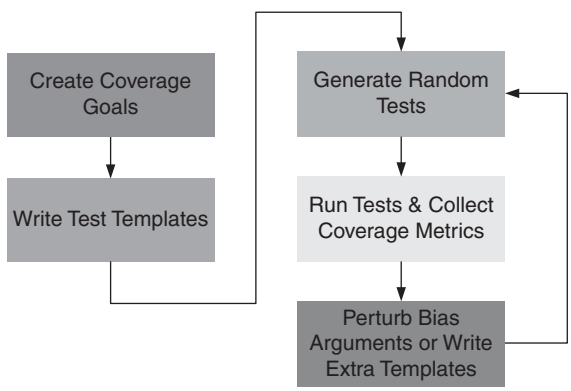



**FIGURE 9.5**

Flow of random testing.

One corresponding template may look like the following:

**MUL** < random R1-R4 >< random R4-R8 >< random R8-R20 >

or

< Pr(**ADD**) = 90% & Pr(**SUB**) = 10% > R3 R5 < random R4-R7 >

In the first template, the instruction is designated to be **MUL** (multiplication), and its three operands can be selected from different registers. In the second example, the actual instruction is decided with a probability, where 90% is to be an **ADD** (addition) and 10% is a **SUB** (subtraction) where the third operand is randomly selected from registers R4 through R7.

Random testing is usually applied at the beginning of the verification process for modern designs. Random tests are applied to randomly exercise the design space that often can cover some nontrivial cases and some corner cases. Advanced constrained random test generation uses architecture knowledge of the design and past experience to better guide the test generation process. Both templates and bias arguments help hide the detailed information from users while still being able to generate legal tests that conform to the architectural constraints of the design.

### 9.3.2 Coverage-driven verification

Storing information during simulation is necessary to identify those scenarios that have been previously verified. Such a task is called **functional coverage analysis**. ***Coverage-driven verification*** (CDV) represents such a method. It measures the current verification progress [James 2003] and then guides the development of new strategies for uncovering any missing features or scenarios.

CDV uses a single test stimulus to explore multiple scenarios automatically. Inheriting the characteristics of random testing, CDV can also discover corner cases that might occur beyond a user's expectations. **Coverage points** such as **assertions** are often placed in the environment to collect data for analysis. After collecting and analyzing the data, the constraints for guiding test generation can be modified, either automatically or manually, to target the missing features or scenarios before the next round of test generation is called. This iterative test generation process is known as ***coverage-directed generation*** (CDG). Figure 9.6 illustrates a typical coverage-driven verification design flow.

CDV [Benjamin 1999; Bergeron 2000; Verisity 2001; Gluska 2003; Palnitkar 2003b] is more effective than constrained random verification and thus achieves verification closure faster. Figure 9.7 illustrates the effectiveness comparison of these two approaches.

Coverage is created to identify the error-prone areas in which bugs may reside. It originates from software testing, which provides a means of assessing the thoroughness of software development. A general definition of coverage is a

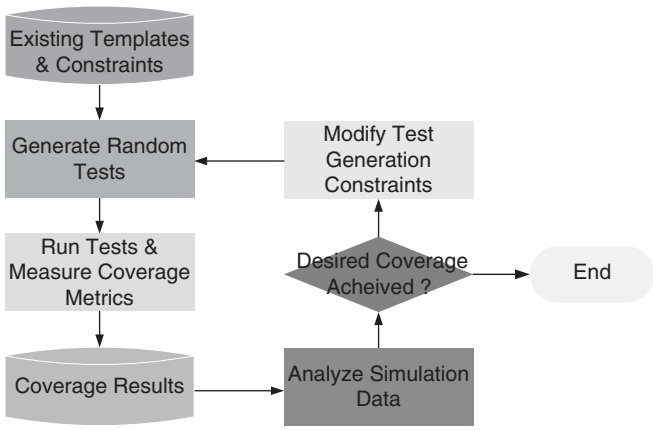



**FIGURE 9.6**

Coverage-driven verification design flow.

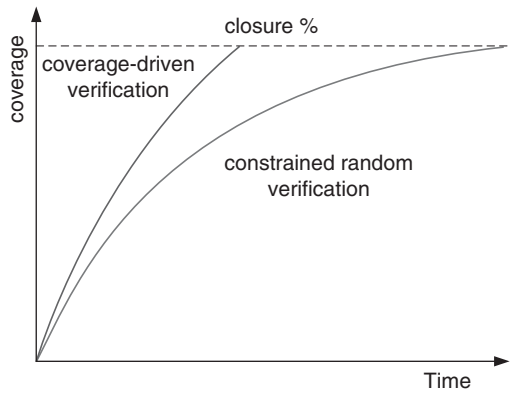



**FIGURE 9.7**

Effectiveness comparison between coverage-driven verification and constrained random verification approaches.

measure of the extent to which the features and scenarios of the design under verification are covered.

Coverage metrics can be classified into two categories—**functional coverage** and **structural coverage**—according to the verification intent. Functional coverage checks the concordance of the **semantic** design intent with the designer's implementation, and it is measured by the number of features and scenarios defined in the design specification that are exercised by the test set. **Structural coverage** aims at measuring the degree of confidence for **syntactic** correctness of the physical implementation that the test set achieves.

### 9.3.3 Structural coverage metrics

Structural coverage measure is also referred to as **code coverage metric,** because the objective is to evaluate whether various kinds of elements in the HDL implementation are exercised by a given test set. Because code coverage metric ties with test vectors and physical representation in the hardware description language, simulation engines can be easily modified to provide the coverage information. Code coverage comes in many forms. The following describes a few among the commonly used metrics.

#### 9.3.3.1 *Line coverage (a.k.a. statement coverage)*

This metric takes the syntactical HDL implementation and counts the number of lines exercised during the simulation run. The line coverage is defined as:

$$\text{Line Coverage} = \frac{\#\text{ of exercised lines in HDL}}{\text{Total}\,\#\text{ of lines in HDL}} \times 100\%$$

Consider the following Verilog HDL code in Box 9.1:

**BOX 9.1**

```
1. always @(in or reset) begin
2.    out = in;
3. if (reset)
4.    out = 0;
5. en = 1;
6. end
```

If the testbench exercises lines 1, 2, 3, 5, and 6, the line coverage would be 5/6 = 83.3%. The line coverage is easy to comprehend, and the missed line explicitly indicates the absence of signal activities. One obvious drawback of line coverage is its lack of a clear connection between the number of exercised lines and the correctness of design intent.

#### 9.3.3.2 *Toggle coverage*

This metric checks whether signals in the design change their values during simulation. It helps verify the quality of the test set and locate the unexercised areas. Signals that fail to be initialized or to toggle by the test cases can be easily identified. Box 9.2 is a sample toggle coverage report.

**BOX 9.2**

```
1. //net toggle coverage
2. //name        Toggle    0→1    1→0
3. clk           Yes
```

| | | | |
|---|---|---|---|
| 4. reset | No | Yes | No |
| 5. start | Yes | | |
| 6. state[6:0] | Yes | | |
| 7. state[9:7] | No | No | No |
| 8. op[2:0] | Yes | | |
| 9. op[3] | No | No | Yes |
| 10. op[4] | Yes | | |
| 11. op[5] | No | No | Yes |
| 12. round[1:0] | Yes | | |
| 13. src1[63:0] | Yes | | |
| 14. src2[63:0] | Yes | | |

Although the toggle coverage is easy to compute, it has similar drawbacks to the line coverage in that it does not provide any insight about the design intent from the toggle events.

#### 9.3.3.3 *Branch/path coverage*

This metric evaluates the control flow, such as *if* and *case*, in RTL statements. It counts the number of branches at decision points that are exercised during simulation. The branch coverage is defined as:

$$\text{Branch Coverage} = \frac{\#\text{ of exercised branches}}{\text{Total}\,\#\text{ of possible branches}} \times 100\%$$

The path coverage refines the branch coverage concept. It does not look at decision points independently. Instead, it considers the whole sequence of decision points, called a path, which could possibly be involved in one clock cycle. Note that when *if* or *case* statements are nested, the total number of possible paths may grow exponentially. Therefore, reaching a 100% path coverage may become difficult.

Consider the preceding exemplar Verilog HDL code in the discussion of line coverage. Assume the signal *reset* is always 1. Then, for the *if* statement, only the *reset* = 1 branch is exercised. Thus, the branch coverage is 1/2 = 50%. Now consider another example:

**BOX 9.3**

```
1. if (x != y)
2.     z = 0;
3. w = z;
```

In Figure 9.8, the RTL code is represented in two flowcharts — each of which is from the line and branch coverage viewpoints, respectively. Assume the values of signal $x$ are never equal to those of $y$ during simulation. Then line 2 will be exercised, resulting in a final line coverage of 100%. But the branch ($x == y$),

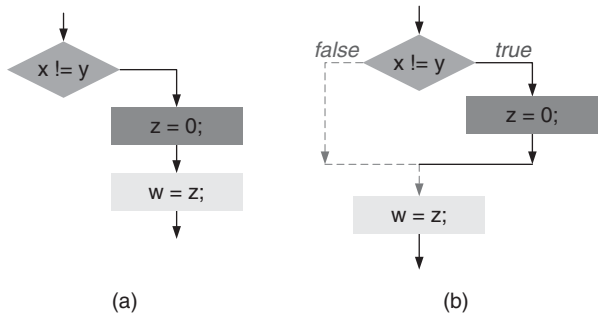



**FIGURE 9.8**

(a) Flowchart for line coverage. (b) Flowchart for branch coverage.

represented by the dotted line in Figure 9.8b, is never exercised, resulting in a branch coverage of only 50%.

Note that designers can implement the branch condition implicitly without the use of *if* or *case* statements. For example, an *if-else* condition can be implemented by a multiplexer that uses AND or AND-NOT operations. Hence, it may not be always apparent to know exactly where to collect the branch statistics to calculate a branch coverage. In many situations, a branch not explicitly implemented by use of *if* or *case* statements may not be accounted for in the coverage.

#### 9.3.3.4 *Expression coverage*

The expression coverage enhances the line and branch coverages and provides more information about concurrent signal assignments. It focuses the analysis on the expression in the right-hand side of an assignment or the expression in a condition statement.

Typically, one expression can be recursively decomposed into multiple **sub-expressions**, which are either a single variable or two variables connected by a logical operator. These sub-expressions are monitored individually during simulation. An expression is fully covered if all of the sub-expressions are exercised. Otherwise, the expression coverage for a line is calculated by deriving the ratio of the total number of exercised cases to the total number of possible cases among all of its sub-expressions.

$$\text{Expression Coverage} = \frac{\Sigma_{i=1}^{k} \#\text{ of exercised cases for sub-expressions } i}{\Sigma_{i=1}^{k} \#\text{ of possible cases for sub-expressions } i} \times 100\%$$

The expression coverage can be further classified into three categories: **multiple sub-condition**, **basic sub-condition**, and **focused expression coverages** [Dempster 2002].

The ***multiple sub-condition coverage*** (MSC) is the most popular and straightforward one. It enumerates all possible combinations of the sub-expressions. That is, if there are $N$ sub-expressions, then $2^N$ cases need to be covered to achieve a 100% multiple sub-condition coverage. Consider the following expression in Box 9.4:

**BOX 9.4**

```
1. if ((A == 0) || ((B == 1) && (C == 0)))
```

The participating sub-expressions are (A == 0), (B == 1), and (C == 0). Thus, the test vectors have to cover all $2^3 = 8$ possible cases to achieve a 100% multiple sub-condition coverage.

The ***basic sub-condition coverage*** (BSC) checks both the true and false states of each sub-expression during simulation. For the preceding example, there are six possible cases: (A == 0) is true, (A == 0) is false, (B == 1) is true, (B == 1) is false, (C == 0) is true, and (C == 0) is false. A sample report, after the basic sub-condition coverage is derived, is listed in Box 9.5:

**BOX 9.5**

| | Count | Sub-expression | Outcome |
|---|---|---|---|
| 1. | Count | Sub-expression | Outcome |
| 2. | 4 | A == 0 | true |
| 3. | 6 | A == 0 | false |
| 4. | 8 | B == 1 | true |
| 5. | 2 | B == 1 | false |
| 6. | 0 | C == 0 | true |
| 7. | 10 | C == 0 | false |

In this report, because the condition "(C == 0) is true" has never been exercised during simulation, the basic sub-condition coverage is (5/6) = 83.33%.

An expression is a function of the participating variables combined with Boolean operators. If one variable in focus can control the result of the expression, there should be a pair of variable assignments for which the values at all other variables, except the focused variable, are the same so that one assignment evaluates the expression to be true and the other assignment to be false. On the basis of this notion, the ***focused expression coverage*** (FEC) is developed, which helps identify the minimum set of tests required for verifying a complicated branching expression. To achieve a 100% FEC for an expression, for each participating variable in the expression, the test set must include a pair of vectors that assign identical values to all other variables except the target variable, and these two vectors evaluate the expression to different values.

To illustrate this notion, consider the expression in Box 9.6:

**BOX 9.6**

```
1. if (A && B)
```

The focused expression coverage criteria for variable A are [A, B] = [0, 1] and [A, B] = [1, 1]. Note that in both cases, B has to be 1 for the effect of changing

A to be observed. Similarly, the criteria for variable B are [A, B] = [1, 0] and [A, B] = [1, 1]. Because [A, B] = [1, 1] is a common assignment, it would require only three assignments to fully validate expression (A && B).

Now consider the following example in Box 9.7:

**BOX 9.7**

```
1. if (((X == 1) && (Y == 0)) || (Z == 0))
```

The three sub-expressions are expr_1 = (X == 1), expr_2 = (Y == 0), and expr_3 = (Z == 0). To achieve a 100% FEC, the test set must include the following tests:

- To target expr_1, [expr_1, expr_2, expr_3] = [0, 1, 0] and [expr_1, expr_2, expr_3] = [1, 1, 0] are required. Note that expr_2 has to be 1 because it is **AND**ed with expr_2. Similarly, expr_3 has to be 0 because it is **OR**ed with the rest of the expression. The result is that (X, Y, Z) = (0, 0, 1) and (1, 0, 1) must be covered.
- To target expr_2, [expr_1, expr_2, expr_3] = [1, 0, 0] and [expr_1, expr_2, expr_3] = [1, 1, 0] are required. Therefore, (X, Y, Z) = (1, 1, 1) and (1, 0, 1) must be covered.
- To target expr_3, there are three different ways to ensure expr_3 controlling the overall expression: [expr_1, expr_2] = [0, 0], [0, 1] and [1, 0] respectively. Therefore, one of following three pairs, (X, Y, Z) = {(0, 1, 1), (0, 1, 0)}, {(0, 0, 1), (0, 0, 0)}, and {(1, 1, 1), (1, 1, 0)} must be included in the test set.

Combining these three requirements, the minimum test set for a 100% FEC includes 4 tests which are either {(0, 0, 1), (1, 0, 1), (1, 1, 1), (0, 0, 0)} or {(0, 0, 1), (1, 0, 1), (1, 1, 1), (1, 1, 0)}.

Suppose a given test set contains only two tests, (X, Y, Z) = (1, 0, 1) and (X, Y, Z) = (1, 0, 0), which evaluate [expr_1, expr_2, expr_3] to [1, 1, 0] and [1, 1, 1], respectively. With respect to the focused expression notion, none of the three sub-expressions is satisfied by these two tests and, thus, its focused expression coverage is 0%.

#### 9.3.3.5 *Trigger coverage (a.k.a. event coverage)*

This metric simply measures the number of exercised variables in the sensitivity list. Consider the example given in Box 9.8:

**BOX 9.8**

```
1. always @(a or b or c)
2. begin
3.    ...
4. end
```

Signals a, b, and c are monitored throughout the simulation. If only b and c change values during simulation, then the trigger coverage would be 2/3 = 66.67%.

#### 9.3.3.6 *Finite state machine (FSM) coverage*

The FSM coverage plays an important role in verifying the control unit of a design. As its name implies, this metric is tied to the HDL structure of finite state machines in a design and can be divided into three sub-classes. The **state coverage** reports the states that are visited and their frequencies during simulation. The **arc coverage** records the state transitions that are traversed during simulation. Even if 100% *state* and *arc coverages* are achieved, there is no guarantee that the FSM is bug-free. Therefore, the third class of FSM coverage, called **sequential arc coverage** (*a.k.a.* **transition coverage)**, was designed. The metric measures the coverage on the basis of an increased sequential depth of state visitation or arc traversal. It also identifies the fundamental cyclic sequences in various lengths. Figure 9.9 shows an FSM example and the arc sequences starting from $s_1$ for calculating the sequential arc coverage. For example, $\{s_1 \rightarrow s_2 \rightarrow s_2\}$ is a *2-arc* transition starting from *s*1 to be monitored for the sequential arc coverage.

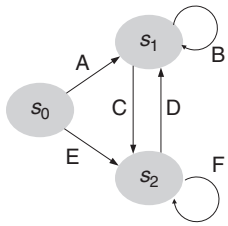
In calculating the coverage, the conventional FSM coverage interprets the RTL code syntactically. That is, it treats each state as a unique state and its state transition to any other state as a unique arc. Although each state has a unique state code, it is common that a group of states have identical or very similar behavior. Therefore, interpreting the FSM syntactically may result in many unnecessary checks. Consider the following partial RTL code of a 4-bit binary counter with *reset* and *load* signals in Box 9.9:

**BOX 9.9**

```
1. always @(posedge clk) begin
2.    if (reset) count = 0;
3.    if (load) count = in;
4.    else if (count == 15)
5.       count = 0;
6.    else
7.       count = count + 1;
8. end
```

The implementation has 16 states. Because any state can go to any other state including itself (either through incrementing the count variable or through loading a new state value in), each state has 16 outgoing arcs, resulting in a total of 256 arcs. Figure 9.10a illustrates this conventional interpretation of the FSM. If the counter is 8-bit, the total number of states will increase to 256 states with 65,526 arcs.

To represent the design as an FSM, it is better to interpret it semantically, which defines the states on the basis of the unique actions taken during the



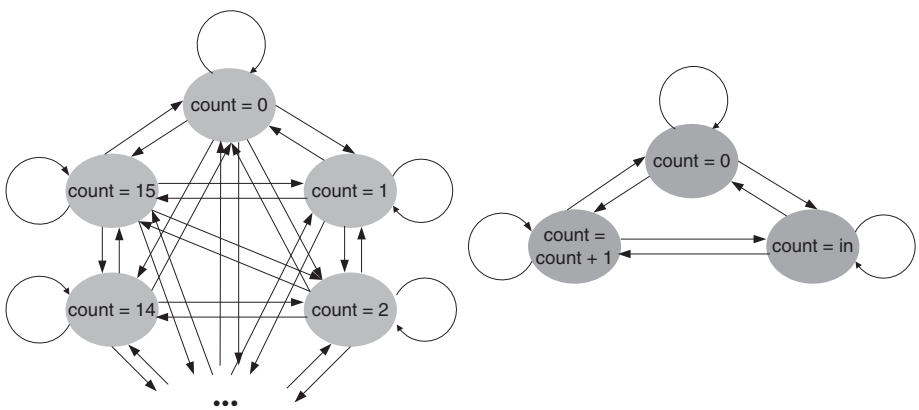

(a)

| length of transition | *1-arc* | *2-arc* |
|---|---|---|
| sequence of states from $s_1$ | $s_1 \rightarrow s_1$ | $s_1 \rightarrow s_1 \rightarrow s_1$ |
| | $s_1 \rightarrow s_2$ | $s_1 \rightarrow s_1 \rightarrow s_2$ |
| | | $s_1 \rightarrow s_2 \rightarrow s_1$ |
| | | $s_1 \rightarrow s_2 \rightarrow s_2$ |

(b)

**FIGURE 9.9**

(a) FSM example. (b) Transition sequences from $s_1$.




(a) (b)

**FIGURE 9.10**

Illustration of (a) the conventional FSM coverage. (b) the semantic FSM coverage.

operation. For the preceding example, there are only three different actions: count = 0, count = in, and count = count + 1. Figure 9.10b shows the FSM of this interpretation, which consists of only three states and nine arcs. The **semantic FSM coverage**, calculated on the basis of this representation, can greatly reduce the number of tests required for achieving a high coverage.

### 9.3.3.7 *More on structural coverage*

Different metrics for structural coverage can be associated with different HDL structures at different design stages. In general, during the behavioral-level design stage, only line, branch, condition, path, trigger, and FSM coverage can be measured. Toggle coverage is often applied to gate-level designs only. The RTL-level design stage has the broadest possible coverage spectrum, and all types of metrics can be applied.

**Table 9.1** Typical Coverage Targets for Different Metrics

| *Metric* | *Coverage Goal (%)* |
|---|---|
| Line | 100 |
| Branch | 100 |
| Condition | 60∼100 |
| Path | >50 |
| Trigger | 100 |
| Toggle | 100 |
| FSM (state and arc) | 100 |

Because these metrics are simple and straightforward, it is often desirable to achieve a high structural coverage. The typical coverage goals for various metrics are listed in Table 9.1 [Dempster 2002].

Even if the desired coverage for these metrics is achieved, it does not guarantee a bug-free design. None of these metrics — or even were we to combine them all — can be guaranteed to cover all the possible erroneous scenarios.

The structural coverage attempts to explore the design space from the implementation perspective. Although the targets of the structural coverage do not necessarily have direct correlation to functional bugs, achieving a high structural coverage can likely increase the chance of bug discovery. A bug may be revealed by a new test that was designed to detect a not-yet-covered structural target.

### 9.3.4 **Functional coverage metrics**

Functional coverage metrics guide test generation and verification from a semantic perspective. They supplement the deficiencies in the code coverage and help improve verification quality. Some companies have stated that functional coverage would be an important component of their next-generation verification methods [Drucker 2002].

Functional coverage metrics usually involve the interpretation of functionality and the related measurements from the specification, and require domain knowledge and instrumentation from the designer and/or verification engineers. Therefore, an automated means of creating functional coverage models does not exist. Typically, verification engineers need to manually develop a list of target functionalities to be verified and to devise different strategies to exercise each case in the list. A functional bug is claimed to be found if the design does not behave as expected with respect to the functional specification after exercising the related verification scenarios.

The verification method based on the functional coverage includes four major tasks:

1. Determining the coverage events to be verified
2. Preparing stimuli to exercise the target events
3. Collecting data from the design under verification
4. Analyzing results to quantify the coverage and identify missing events

Basically, it is the designer's job to determine the functions to be covered. Verification engineers are required to create a verification plan on the basis of their understanding of the design's functional specification. In addition to enumerating the functions under verification, external resource expenditures, including verification time, manpower, and related software and tool costs, should also be carefully considered.

The verification plan forms the basis for developing the corresponding test programs. Random testing techniques are often used at the transaction level to facilitate test program development. For the AMBA APB part of the example in Figure 9.4, transactions considered for functional coverage could be based on either a simple operation, like a Read/Write to RAM, or a complicated operation, like a sequence of back-to-back Reads to the same address in RAM.

## 9.4 SIMULATION-BASED APPROACH

[Bergeron 2000] introduced a *re-convergence model* for the general design and verification process. Figure 9.11 illustrates the application of this model to functional verification. The designer's effort is dedicated to transforming the functional specification into an implementation in HDL, whereas the verification effort ensures that the transformation is as intended without misinterpreting any functionality.

The functional verification process is typically associated with the concept of *testbench,* which refers to the environment used to apply the predetermined sequence of input vectors to the ***design under verification*** (DUV) and to observe the responses. Figure 9.12 illustrates a DUV surrounded by a testbench. No external communication is required in this system. The testbench models certain aspects of the design intent and is responsible for delivering the input sequences to the DUV and for receiving the output responses for subsequent analysis.

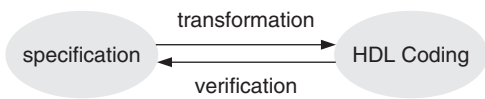



**FIGURE 9.11**

Re-convergence model for the design and verification process.

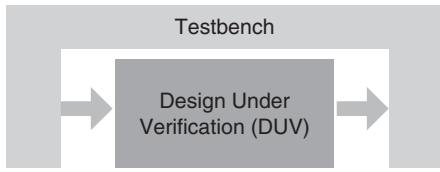



**FIGURE 9.12**

Generic structure of the design under verification and its testbench.

### 9.4.1 Testbench and simulation environment development

In general, the testbench is an HDL description used to create a closed system on top of the design under verification. A testbench consists of three fundamental components: a **stimuli driver**, a **monitor**, and a **checker**.

The **stimuli driver** is responsible for providing stimuli to the DUV. The stimuli can be either predetermined or generated during simulation. The purpose of the stimuli driver is not to mimic the behavior of the entire neighboring blocks but to maintain the interface coherence to the DUV.

The **monitor** is used to observe signal at the inputs, outputs, and any internal wires of interest on the DUV. The values at the input and output signals must be consistent with the interface protocol, and the monitor will issue an error if any exception occurs.

A **checker** can be viewed as a special type of monitor for checking the functionality of the design intent. Traditionally, designers create the functionality checkers manually and use them to compare the responses from the design with the specification. As designs become more complicated, the need to automate the development of such checkers increases.

On the basis of the coverage metrics, verification engineers try to prepare a set of test cases to cover the target functional events. In developing such test cases, experience plays a crucial role. Creating meaningful test cases for some specific events often rely heavily on a designer's knowledge and interpretation of the specifications.

Consider a 16-bit one-hot encoding bus protocol. To achieve an optimal coverage for all scenarios, the test cases would require each bit taking a turn to be 1 with others being 0. In deriving the test cases, it could be difficult to observe the regularity solely from the structure of a design implementation. However, having knowledge of the functionality of the protocol would help capture the regularity and similarity for each bit that make test generation easier and more efficient.

Enumerating deterministic test cases to cover all functions is tedious. An alternative is to convert a design specification into an HDL model to automate the checking. Such a testbench is called a *self-checking* testbench, because checking instrumentation is no longer needed. The *self-checking* testbench

paradigms can be divided into three types: **checking with golden vectors**, **checking against a reference model**, and **transaction-based checking**.

**Checking with golden vectors** is the most widely used approach among the three. Given coverage metrics, the verification engineers search for test cases at inputs and derive the corresponding output responses manually or by use of an auxiliary program. Such combinations of input and output vectors are called the golden vectors. After the testbench applies the input vectors to the DUV, the actual responses are captured and compared with the golden vectors. A bug is found when a mismatch occurs between the golden and the actual responses. Figure 9.13 shows the components of this method.

The **checking-against-a-reference-model** paradigm uses a reference model that captures all functions in the specification. The reference model is typically implemented at a more abstract level with either a high-level programming or a verification language. All input vectors are applied to both the reference model and the DUV, and their responses are evaluated and compared. If the comparison takes place at the end of each cycle, the reference model must be cycle-accurate. The checker compares the responses from both the DUV and the reference model, as illustrated in Figure 9.14. If the specifications change, the reference model would need to be modified accordingly. This modification effort is usually much lower than the effort of reproducing all golden vectors required for the *checking-with-golden-vectors* paradigm.

**Transaction-based checking** is applicable to the DUV that can correspond to commands and data in a transaction. It uses a scoreboard to record the verified command and data. The checker is used to query the scoreboard. It issues an error if the identifier cannot match any transaction in the scoreboard or if the

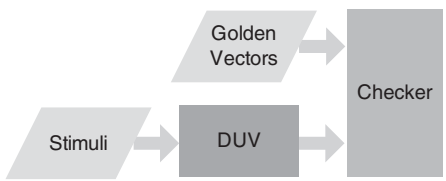



**FIGURE 9.13**

Self-checking testbench with golden vectors.

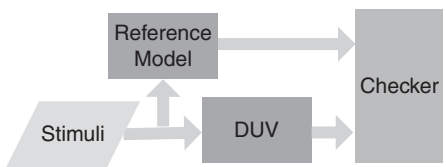



**FIGURE 9.14**

Self-checking testbench with a reference model.

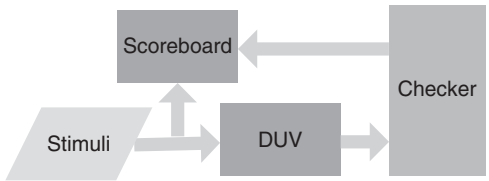



**FIGURE 9.15**

Transaction-based self-checking testbench.

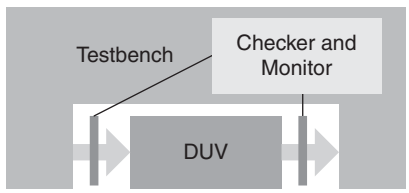



**FIGURE 9.16**

Black-box verification.

command and data are not the expected values given by the scoreboard. This concept is illustrated in Figure 9.15.

## 9.4.2 Methods of observation points

As we can see in the preceding, the monitor and checker in one testbench are tightly tied to the concept of observation of signal changes in the DUV. Such observation approaches will also determine the strategy used for generating stimuli. The three common verification paradigms regarding the observation points are the **black-box**, the **white-box**, and the **grey-box** methods.

The **black-box** method assumes the internal signals of the DUV are not accessible during verification. Only the external input/output interfaces are directly controllable and observable. The verification plan, including the testbench development, is developed based only on input/output functionality. Figure 9.16 illustrates this method.

The major advantages of black-box verification are its simplicity and independence from specific implementation information. Of all the verification methods, it requires the least amount of knowledge about the DUV. Even if the design's HDL code is not ready, the verification process can be started, and stimuli can be developed as long as a reliable specification for the DUV becomes available. Whether the DUV is realized as an ASIC, an FPGA, a circuit board, or a software program is irrelevant. The black-box method only aims at verifying the functionality defined with respect to the design boundaries.

On the other hand, without any structural information, black-box verification lacks the observability and controllability internal to the DUV, which sometimes might be required to determine whether the DUV passes or fails a specific test. It is challenging to precisely identify what and where a problem is in the DUV with this method. It may not be feasible for the black-box verification to check for DUV's low-level features and structural changes. Black-box verification may not be suitable for designer-level blocks, because many interesting corner cases may be observed only when implementation details are provided.

In short, the black-box method requires no implementation knowledge and demands only design specification to complete the testbench development. Being independent of the implementation makes the generated stimuli more reusable for different realizations, but it also makes the stimuli generation process more difficult because of the lack of observability and controllability internal to the DUV.

The **white-box** method, which is illustrated in Figure 9.17, represents another extreme scenario. Here, the full observability and controllability internal to the DUV is assumed to be available. For controllability, verification engineers can easily derive stimuli for the desired events by setting up the required internal states and justifying these states backward toward the inputs. Likewise, regarding observability, any changes in internal signals can be directly observed. Therefore, the white-box method can pinpoint the problematic area in the DUV once a mismatch from the expected value is observed.

Low-level features and implementation changes can be incorporated in the white-box approach, because such verification is tied to a specific implementation. Therefore, the generated test cases may only be valid for the specific implementation. Modification to the generated test cases would be necessary if the implementation changes. Therefore, the maintenance efforts required for the white-box method would be much greater than those for the black-box method.

White-box verification can ensure that implementation-dependent features are verified. For example, it becomes feasible to generate test cases to exercise a timing-critical path when the full observability and controllability to the internal structure of the DUV is available.

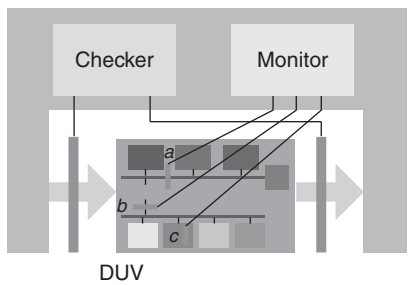



**FIGURE 9.17**

White-box verification.

The **grey-box** approach is a compromise between the black-box and the white-box approaches, which inherits the advantages from both methods. This approach intends to exercise only those significant features associated with the implementation.

The general architecture of the DUV is assumed to be known by the verification engineers, and only a limited number of internal points are accessible. These observation points are often located in the inter-block interface and adhere to specific communication protocols. In other words, the grey-box verification method observes only a select set of important internal signals, which are typically located at the boundary of a building block. Therefore, for the illustration in Figure 9.17, a grey-box method would preclude observation of the monitor ***c*** but would include the other two observation points.

Similar to the white-box approach, the grey-box approach could exercise a desired event by applying a test case directly at inter-block interfaces. Even if the implementation of the components changes, as long as the interfaces between the components within the DUV remain unchanged, the generated test cases can be reused.

### 9.4.3 **Assertion-based verification**

Assertion-based verification is becoming popular in the industry and has drawn much attention in the recent literature [Foster 2004]. This method embeds a set of assertions in various parts of the implementation for monitoring design properties. Assertion-based verification can be viewed as a variant of the white-box method.

The concept of assertions is originated from software testing. An assertion is a line in the program that checks the validity of an expression. A correct program must guarantee that such expressions are always true; otherwise, a warning or exit signal should be issued. Software engineers frequently write assertions to check the possible existence of unexpected scenarios. Many high-level programming languages such as C/C++, Java, and Eiffel support assertions by the use of a system library or by the use of the language definition itself. Actually, the first standardization of VHDL defined its language constructs to support simple assertions, as shown in Figure 9.18.

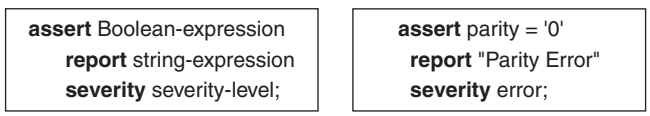



**FIGURE 9.18**

(a) Syntax. (b) example of an even-parity assertion in VHDL.

Similar to software testing, assertions in hardware design are also expressed as part of the design description in the HDL code. Many contemporary ***hardware verification languages*** (HVLs), such as **SystemVerilog** [Accellera 2002a] and **OpenVera** [Synopsys 2001], were developed to facilitate the writing of assertions in conjunction with the design itself. Another flavor of practical solutions is to use an auxiliary specification language. Several different proprietary formats of specification languages exist, such as **PSL/Sugar** [Accellera 2002b]. Assertions can be written in the specification language with a proper interface to the design.

The use of assertions in verification has various advantages. In black-box verification, for example, assertions can be used to replace the original monitors for the purpose of collecting coverage data. In white-box verification, the origin of an assertion failure could be confined to a limited area to facilitate the debugging process. It is also a good practice to use assertions as formal comments in place of comments in natural language. Meanwhile, assertions can be reused as part of the verification IP associated with the IP core delivered to the customers. Moreover, because assertions are placed in the HDL code, they can be directly used as properties to be checked for the use of formal methods.

#### 9.4.3.1 *Assertion coverage and classification*

The term ***assertion coverage*** has a variety of definitions. It could be used to indicate the ratio of the number of assertions to the number of HDL code lines. However, **assertion density**, suggested in [Piziali 2004], is considered a better term for this definition. The better definition for assertion coverage should be similar to that of functional coverage, which is defined on the basis of the number of exercised scenarios over the total number of scenarios to be covered. ***Assertion coverage*** counts the number of exercised assertions to the total number of assertions extracted from the design implementation.

Assertions can be classified into two types: *static* and *temporal*.

- **Static assertions** dictate those legal scenarios that are not related to time, and, as such, they are required to be held for all time. These scenarios can be described by the first-order logic. The one-hot encoding bus is an example. Only one bit in such a bus can be one, and the rest should be zero. A static assertion monitors the bus during the course of simulation and sends an error message whenever this rule is violated.
- **Temporal assertions** extend the capability of static assertions to temporal logic. The consequent statement needs to be evaluated during the specified period of time after which the antecedent condition is triggered. Consider the following SystemVerilog example in Box 9.10:

**BOX 9.10**

```
1. @(posedge clk)
2.    init_event |=> abort_event;
```

where |=> denotes the non-overlapping implication operator. This example states that once an antecedent condition, **init_event**, successfully completes, a consequent statement, **abort_event**, will occur in the next clock cycle.

The behavior of temporal assertions can be illustrated by a finite state machine, as shown in Figure 9.19. In the *idle* state, the assertion moves to the *evaluate* state when its antecedent condition is triggered. The *evaluate* state repeatedly checks the consequent statement before a Pass/Fail result is issued. Once there is a result, either an error signal is generated or the system moves back to the *idle* state.

To illustrate a ***SystemVerilog Assertion*** (SVA) example, assume that the intended property in a design is the following: "after the request signal is asserted, the acknowledge signal must be generated from 1 to 3 cycles later." Figure 9.20 shows its timing diagram and the corresponding code in SVA.

### 9.4.3.2 *Use of assertions*

For different types of properties, assertions can be divided into two categories: **coverage assertions** and **checker assertions**. Coverage assertions primarily record the occurrence frequency of a specified event. Such assertions usually monitor events defined in the functional coverage metrics. For the example of a 16-bit one-hot coded bus, the assertion defines all possible combinations of 16 one-hot cases and records the case(s) exercised during simulation.

Checker assertions function as sentinels. They watch the violation of static or temporal properties. At the module level, in white-box verification, assertions

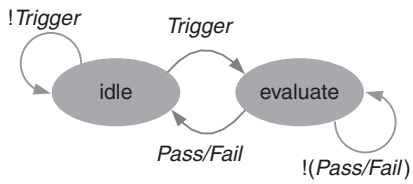



**FIGURE 9.19**

Finite state machine for generic assertions.

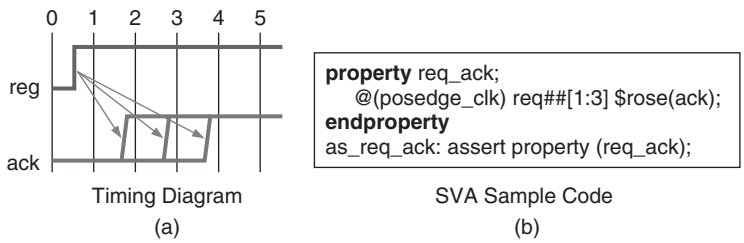



**FIGURE 9.20**

Example of a temporal assertion in SVA.

can check implementation details, whereas in black-box verification, assertions check against the specification through both module inputs and outputs. For higher-level verification, checker assertions are used to monitor the interfaces across components. Because the interfaces must abide by their corresponding protocols, checker assertions signal errors once unexpected scenarios occur. A two-hot message in a one-hot coded bus is such an example.

#### 9.4.3.3 *Writing assertions*

One of the most frequently asked questions in assertion-based verification is "Who should write the assertions?" In practice, this job is shared by the entire design and verification team. At different levels of the design abstraction, different properties are converted into assertions. It may be difficult to ask a designer responsible for designing a small block and lacking a system-level view to write high-level assertions.

At the architectural level, a design is described by use of the input/output functions of each component and the interface protocols that connect them, without implementation detail. Assertions at this level model high-level relationships and ensure that system-level behavior is consistent with the system-level specification. Also at this level, observation points are located at inputs and outputs of the components and at bus interfaces only.

Assertions try to capture one's understanding of the design intent. Once a design component is created, the designer can write assertions for it on the basis of the functionality from the specification and the implementation he or she chooses. At this level, assertions are frequently used for debugging and for measuring coverage.

If applicable, verification engineers may use formal methods to prove assertions to complement the deficiencies of simulation-based methods. Also, assertions accompanied with IP cores from IP providers would need to be integrated into the verification plan.

## 9.5 FORMAL APPROACHES

Advances in modern simulators allow full-chip simulation to be efficiently conducted. Nevertheless, the success of simulation-based verification remains dependent largely on the quality of the stimuli. The stimuli exercise a ***design under verification*** (DUV) and traverse its state space. Verification can be considered as a process of exploring reachable state space of the design. Modern designs rapidly increase in size and complexity, and, consequently, their reachable state space can grow exponentially. As a result, it becomes difficult to exhaust all reachable states for complete verification by use of only simulation.

Formal approaches aim to make complete verification possible, where completeness is in the sense that all reachable states are explored. The underlying idea is to infer the design properties by reasoning without explicitly simulating

stimuli. A property models certain aspects of design behavior associated with all or a subset of reachable states. Proving design properties with formal approaches requires the use of efficient search or reasoning engines, many of which have been developed over the years. Significant advances have been achieved in recent years.

The remainder of this chapter provides an overview of modern formal verification approaches. Three major types of formal approaches are introduced: *model checking*, *equivalence checking,* and *theorem proving*. For each approach, we explain the underlying theory, illustrate its use, give examples, and discuss the advantages and disadvantages. Finally, we include a brief review of advanced research topics in the area.

### 9.5.1 Equivalence checking

Modern VLSI design flow is partitioned into a number of synthesis steps that take the idea from system specification into GDSII. This results in descriptions at different abstraction levels, which include behavioral, RTL, gate, and switch levels. Ensuring equivalence between two alternative descriptions of the same design is a commonly encountered problem in a design process. This task is referred to as *equivalence checking*. Although such a general concept can be applied to detect any mismatch from two descriptions given at any level, commercially available equivalence checking tools typically address the equivalence between the design's RTL code and its various gate-level netlists, as shown in Figure 9.21. That is the focus of this section.

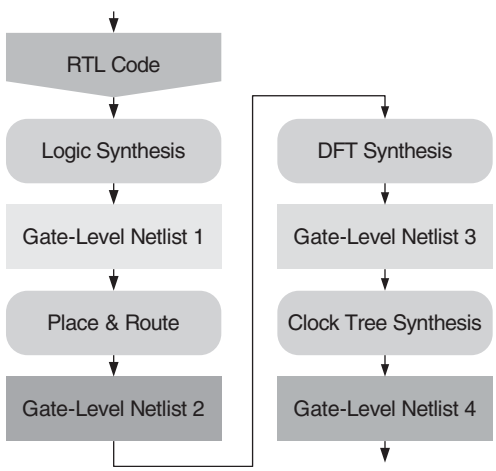



**FIGURE 9.21**

RTL to gate-level design flow.

Boolean circuits, in general, can be viewed as ***finite state machines*** (FSMs), and, therefore, ***Boolean equivalence checking*** (BEC) over two circuits, *FSM₁ and FSM₂*, can be formulated as the problem of checking for the output of the **miter circuit**, as shown in Figure 9.22, being constant 0 or not. $FSM_1$ consists of combinational logic $C_1$ and a state-holding element set, $S_1$, whereas $FSM_2$ consists of combinational logic $C_2$ and a state-holding element set, $S_2$. Both primary inputs are $m$ bits and primary outputs are $n$ bits. $PPO_1$ ($PPO_2$) denotes the pseudo-primary outputs from to $C_1$ ($C_2$) to $S_1$ ($S_2$). Note that the number of state-holding elements can be different in the two FSMs. Each pair of corresponding primary output bits — one from $C_1$ and the other from $C_2$ — connects to an XOR gate. If any XOR output becomes 1 with respect to any input vector or sequence, these two FSMs are not equivalent.

A simplified version of the BEC problem is ***combinational equivalence checking*** (CEC). This problem assumes that $FSM_1$ and $FSM_2$ have a complete, one-to-one mapping between the state-holding elements and that they start with the same initial state. The assumption is also made that $PPO_1$ always has the same value as $PPO_2$. Hence, the original miter circuit can be recast as that shown in Figure 9.23; here, we only focus on the comparison between combinational logic $C_1$ and $C_2$ without any sequential elements. The combinational equivalence checking problem is thus formulated as the following: Given two combinational Boolean netlists $C_1$ and $C_2$, check whether the corresponding outputs of $C_1$ and $C_2$ are equal for all input combinations. There are two types of approaches for solving the CEC problem: *functional equivalence* and *structural equivalence*.

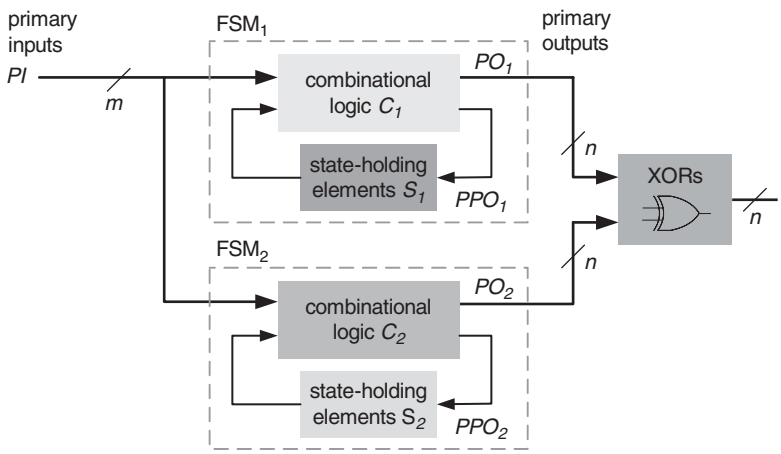



**FIGURE 9.22**

Miter circuit for checking equivalence of two FSMs.

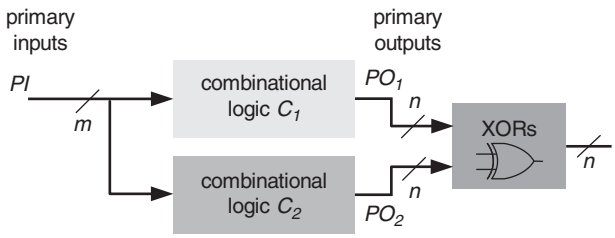



**FIGURE 9.23**

Combinational equivalence checking.

### 9.5.1.1 *Checking based on functional equivalence*

The first step of functional CEC is to translate the combinational circuits into a *canonical representation*. A representation of a Boolean function is *canonical* if the representation for each function is unique and independent of the implementation of the function. A *truth table* is one example of a canonical representation for Boolean functions. Equivalence can be determined by directly comparing the two canonical representations. Among all canonical representations, the reduced ***ordered binary decision diagram*** (OBDD), introduced in Chapter 4, is the most prevalent, because OBDD yields a more compact representation than other representations. The CEC problem can be resolved by building the OBDDs for the outputs of the circuits on the basis of their primary inputs. Two circuits are equivalent if the OBDDs from each pair of corresponding outputs are graphically isomorphic.

### 9.5.1.2 *Checking based on structural search*

A structural search approach checks to see whether any vector exists at primary inputs that would cause a mismatch between the two circuits at their primary outputs. If no such input vector can be found, the two circuits are proven equivalent. The ***satisfiability*** (SAT) solvers, introduced in Chapter 4, can be used as the structural search engine for checking equivalence. A SAT solver can be used to check if an assignment at PIs exists to satisfy a 1 at the miter's output. An UNSAT answer from the solver proves the equivalence of the two circuits. An ATPG tool developed for generating manufacturing tests for stuck-at faults can also be used for checking structural equivalence. As illustrated in Figure 9.24, if the stuck-at-0 fault at the XOR output is proven a redundant fault by an ATPG tool, the two circuits are equivalent. A thorough treatment of ATPG techniques will be provided in Chapter 14.

For complex circuits, directly applying SAT solving at the miter's output signal may result in an exponential number of backtracks, which makes the approach inefficient. Structural similarity between the two circuits under checking can be explored to improve its efficiency, which attempts to solve the structural equivalence problem by incrementally solving a sequence of easier sub-problems

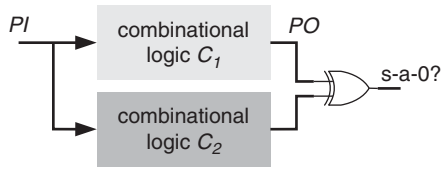



**FIGURE 9.24**

Checking structural inequivalence by generating a test for XOR output stuck-at-0 fault.

[Brand 1993; Kunz 1993; Goldberg 2000; Huang 2000]. On the basis of a divide-and-conquer strategy, various heuristics have been developed to identify internal equivalent points from the two circuits under checking. For example, when two signals are proved to be equivalent, the equivalence of the two signals can be encoded as a SAT clause and added back to the SAT formulation of the problem. Such equivalence clauses can then help to speed up the SAT search, as shown in [Lu 2003].

For the ***sequential equivalence checking*** (SEC) problem, shown in Figure 9.22, *state traversal* techniques are often used. The most common state traversal technique is *reachability analysis*. Note that two FSMs, $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$, are equivalent if, and only if, the output of the *miter circuit* $\boldsymbol{M}_{1\otimes 2}$ is constant 0 under all combinations of input assignments for all *reachable states* of $\boldsymbol{M}_{1\otimes 2}$. Therefore, checking sequential equivalence would require the ability of deriving the set of states reachable from a given initial state set $\boldsymbol{I}$ for a given FSM $\boldsymbol{M}$. An intuitive approach that explicitly enumerates state transitions over the state graph of the FSM is not scalable to large design and, thus, is often impractical. Practical solutions usually adopt a *symbolic* technique implemented by OBDD that implicitly derives the reachable state set by use of *transition functions*.

**Symbolic reachable analysis** consists of two steps: (1) encoding the FSM symbolically and (2) performing reachability analysis iteratively. Given FSM $\boldsymbol{M}_1 = (Q_1, I_1, \sum_1, \Omega_1, \delta_1, \lambda_1)$ and FSM $\boldsymbol{M}_2 = (Q_2, I_2, \sum_2, \Omega_2, \delta_2, \lambda_2)$, where $Q_i$'s, $I_i$'s, $\sum_i$'s, $\Omega_i$'s, $\delta_i$'s, $\lambda_i$'s denote the state spaces, the initial state sets, the input and output alphabets, transition functions, and output functions, respectively, the FSM $\boldsymbol{M}_{1\otimes 2} = (Q_m, I_m, \sum_m, \Omega_m, \delta_m, \lambda_m)$ for the miter circuit can be constructed as follows:

- The state space $Q_m = Q_1 \times Q_2$
- The initial state set $I_m = I_1 \times I_2$
- $\sum_m$ and $\Omega_m$ are the same input and output alphabet sets as in $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ (that is, $\sum_m = \sum_1 = \sum_2$ and $\Omega_m = \Omega_1 = \Omega_2$)
- The transition function $\delta_m(s, a)$: $\sum_m \times Q_m \rightarrow Q_m$, where $s$ and $a$ represent for one state in $Q_m$ and one input vector in $\Sigma$, respectively
- The output function $\lambda_m(s, x)$: $\sum_m \times Q_m \rightarrow \Omega_m$

We define a new function, called *transition relation*, which is denoted as $R(x, s, s')$: $(\sum_m \times Q_m) \times Q_m \rightarrow \{0, 1\}$. $R(a, p, q) = 1$ if there exists a transition from the state $p$ to the state $q$ under an input vector $a$ for $\boldsymbol{M}_{1\otimes 2}$; otherwise, $R(a, p, q) = 0$. Assume

given an input vector set $x = (x_1, x_2, \ldots, x_k)$ with the corresponding sequence of state transitions $\delta_x = (\delta_1, \delta_2, \ldots, \delta_k)$, the transition relation from the state $s$ to the state $s'$ can be formulated as:

$$R(x, s, s') = (s_1' \equiv \delta_1(s, x)) \wedge (s_2' \equiv \delta_2(s, x)) \wedge \ldots \wedge (s_k' \equiv \delta_k(s, x)) = \Pi_i(s_i' \equiv \delta_i(s, x))$$

Therefore, if the input vector set $x$ can bring the finite state machine from the state $s$ to the state $s'$, then $R(x, s, s') = 1$; otherwise, $R(x, s, s') = 0$.

We then annotate the existential quantification operator $\exists$ to the transition relation $R$. A pair of states $(p, q) \in R_\exists$ if, and only if, there exists an input vector $x$ such that the machine transitions from state $p$ to state $q$ after applying $x$. Applying the *existential quantification* notation $\exists$ to the preceding *transition relation* results in $R_\exists(s, s')$. Such a notation is called **quantified transition relation** and represented as:

$$\begin{aligned} R_\exists(s, s') &= \exists x.(s_1' \equiv \delta_1(s, x)) \wedge (s_2' \equiv \delta_2(s, x)) \wedge \ldots \wedge (s_k' \equiv \delta_k(s, x))) \\ &= \exists x.\ \Pi_i(s_i' \equiv \delta_i(s, x)) \end{aligned}$$

Given $M_{1\otimes 2} = (Q_m, I_m, \sum_m, \Omega_m, \delta_m, \lambda_m)$ and its *quantified transition relation*, we can apply $R_\exists$ to derive all reachable states. Such a process is called reachability analysis and can be done by the **image computation** denoted as $Img(S, R_\exists)$, where $S$ is a set of given states and $R_\exists$ is the *quantified transition relation* defined by $M_{1\otimes 2}$. The output of $Img(S, R_\exists)$ is the set of states reachable from $S$ in one clock cycle. One approach to reachability analysis is to iteratively perform image computation starting from the initial state set $I_m$. Such an approach is called *forward* reachability analysis, and the generic pseudocode is outlined as follows:

**Algorithm 9.1** Forward_Reachability

```
1. i:= 0                                  // counter for looping
2. Q^i:= I                                // i-th set of reachable states
3. do {
4.      Q_new:= Img(Q^i, R_∃);            // compute image from current states
5.      Q^(i+1) := Q^i ∨ Q_new;           // update the state set for next iteration
6.      i:= i + 1;                        // counter increments
7. } until (Q^(i+1) ≡ Q^i)                // stop when state set is stable
8.
9. return Q^(i+1)
```

Consider the 7-state FSM shown in Figure 9.25 for which state 0 is the only initial state. The forward reachability algorithm derives all reachable states from state 0 as follows in Table 9.2:

The iterative process stops at iteration 4 for which the current set of reachable states is equivalent to the next set of reachable states. Therefore, the set of reachable states from state 0 is {0, 1, 2, 3}. From this analysis, we find that states

**Table 9.2** Reachable States by Forward Reachability Algorithm

| Iteration | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $Q^j$ | {0} | {0, 1, 2} | {0, 1, 2, 3} | {0, 1, 2, 3} |
| $Q_{new}$ | {0} | {1, 2} | {1, 3} | {0, 1, 3} |
| $Q^{j+1}$ | {1, 2} | {1, 3} | {0, 1, 3} | {0, 1, 2, 3} |

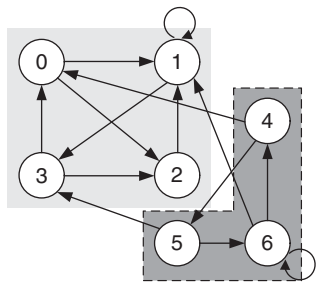



**FIGURE 9.25**

Example of forward reachability analysis.

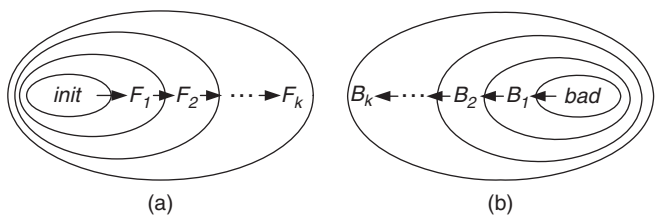



**FIGURE 9.26**

Intuitions behind forward and backward reachability analysis.

4, 5, and 6, which are surrounded by the dotted line in Figure 9.25, can never be reached from the initial state 0. These states form the set of unreachable states for state 0.

Reachability analysis can be also conducted through a *background* traversal of the state space [Abdulla 2000]. For a target final state (which could be a state that causes non-equivalence of the two FSMs), the search attempts to compute the set of previous states that can transition into this target state. If the backward reachability analysis can eventually reach an initial state, the search stops, and the two FSMs are proven not equivalent. Intuitions behind the forward and backward reachability analysis are illustrated in Figure 9.26a and Figure 9.26b, respectively.

Image computation may suffer from too many iterations and/or memory explosion. Several techniques that attempt to avoid memory explosion, such as the use of SAT solving instead of BDD-based techniques [Abdulla 2000], have been proposed.

*Boolean equivalence checking* has been widely accepted and incorporated into industrial design flows. Most leading EDA vendors offer BEC tools that include Encounter Conformal from Cadence and Formality from Synopsys. Combinational equivalence checkers have enjoyed tremendous success, partially thanks to the recent advances in SAT solving, which help to improve both performance and scalability of CECs. Sequential equivalence checking has also made significant progress in recent years. SEC tools such as SLEC from Calypto [Calypto 2008] are also commercially available.

### 9.5.2 Model checking (property checking)

Given a property and a design, a model checking tool allows a user to check whether the property holds true on the design. To develop such a tool, one needs to ask two basic questions: how to specify or describe a property and how to efficiently prove that a property holds true or is violated. The first question concerns the language used to express properties. Such a language determines what properties can be described and what properties cannot be described and, hence, limits the applicability of a model checking tool. The second question concerns the computation engine used to prove properties. Like equivalence checking described previously, OBDD and SAT are two prevalent methods that are used to implement the core computation engine of a model-checking tool. In this section, we begin by introducing the (formal) languages used to describe properties, followed by a brief review of how OBDD and SAT can be used to implement a model checking tool.

**Temporal logic**, introduced by Arthur Prior in 1960s [Prior 1957] and initially known as **Tense Logic**, provides a formal system for qualitatively describing and inferring how the values of statements for properties vary over *time* in a system. In temporal logic, a statement's truth value can change over time. In contrast, in traditional predicate logic, a statement's truth value is either true or false, which does not change over time. Application of temporal logic in verification started to receive attention in 1980s.

Temporal logic consists of two types of formulas: (1) *state formulas,* a form of ***atomic propositions*** (AP ) that indicate the validity of specific states; and (2) *path formulas*, in which the property of a path holds constant. Note that a path here refers to a sequence of states. According to the views taken with respect to the underlying nature of time, temporal logic can be classified into (1) ***linear temporal logic*** (LTL), where the future value can only be derived along its linear computation path; and (2) ***branching time temporal logic*** (BTTL), which is a tree-like structure that allows quantifications over many different futures at each moment. Whether LTL or BTTL is more suitable for model checking depends on the property and the design being checking [Emerson 1990].

LTL allows applications to reason about the nondeterministic behavior. It models time as a sequence of discrete states starting from an initial moment with no predecessors and extending infinitely into the future. Such a sequence

of states is known as either a computation path or an execution path. LTL derives the change over time with a linear time model $\boldsymbol{M} = (\boldsymbol{S}, \rightarrow, \boldsymbol{L})$, which is also known as a **Kripke** structure [Kripke 1963]. Here,

$\boldsymbol{S}$: a set of state formulas $\{s_0, s_1, \ldots\}$
$\rightarrow$: the transition relation where $\forall s \in \boldsymbol{S}$, $\exists s' \in \boldsymbol{S}$, s.t. $s \rightarrow s'$
$\boldsymbol{L}$: a labeling function $\boldsymbol{L}$:$\boldsymbol{S} \rightarrow$P($AP$) in which each state is labeled with a set of atomic propositions from $AP$.

Figure 9.27 shows a simple example of a linear time model, $\boldsymbol{M}_1$, where

$\boldsymbol{S} = \{s_0, s_1, s_2, s_3\}$
$\rightarrow = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_3), (s_2, s_3), (s_3, s_0), (s_3, s_3)\}$
$\boldsymbol{L} = \{(s_0, \{p,q\}), (s_1, \{r,t\}), (s_2, \{q,t\}), (s_3, \{r\})\}$

A path $\pi$ in $\boldsymbol{M} = (\boldsymbol{S}, \rightarrow, \boldsymbol{L})$ is an infinite sequence of ordered states $\{s_i \in \boldsymbol{S}\}$ such that for each $i \geq 1$, $s_i \rightarrow s_{i+1}$. Therefore, path $\pi$ can be expressed as $\pi = \{s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_i \rightarrow \ldots\}$. Particularly, $\pi^k$ denotes the suffix of a path starting from the $k^{\text{-th}}$ state. For example, $\pi^3 = \{s_3 \rightarrow s_4 \rightarrow \ldots\}$. The notations $\models$ and $\not\models$ denote the *satisfaction* relation and the *unsatisfaction* relation, respectively. Given a *Kripke* structure $\boldsymbol{M} = (\boldsymbol{S}, \rightarrow, \boldsymbol{L})$, $\pi \models \phi$ denotes that the formula $\phi$ holds true (*i.e.,* is satisfied by the system) at the starting point of the path $\pi$ in $\boldsymbol{M}$. Let $I(s_1)$ be the set of formulae that hold true at the starting point of path $\pi$. Then, "$\pi \models \phi$" means "$\phi \in I(s_1)$."

LTL is built up from a set of propositional variables $p_1, p_2, \ldots, \top$(true) and $\bot$ (false), the usual logic connectives $\neg$(negation), $\vee$(disjunction), $\wedge$(conjunction), $\rightarrow$(imply), and the following temporal modal operators: **X**(*Next*), **G**(*Always*), **F**(*Finally*), **U**(*Until*), and **R**(*Release*):

- *Next* (**X**) operator is unary and specifies that a formula holds at the *second* state on the path $\pi$:

  $\pi \models \mathbf{X}\phi$ *iff* $\pi^2 \models \phi$

- *Always* (**G**) operator is unary and specifies that a formula holds along *every* state on the path $\pi$:

  $\pi \models \mathbf{G}\phi$ *iff* $\forall i \geq 1, \pi^i \models \phi$

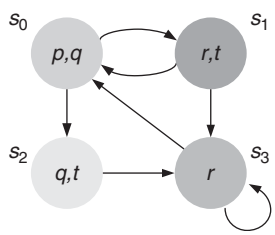



**FIGURE 9.27**

Example of an LTL model.

- *Finally* (**F**) operator is unary and specifies that a formula holds at *some* future state on the path $\pi$:

$$\pi \models \mathbf{F}\phi \ \textit{iff} \ \exists i \geq 1, \pi^i \models \phi$$

- *Until* (**U**) operator is binary and specifies that for some $i \geq 1$, $\pi^0$ to $\pi^{k-1}$ satisfies the first formula $\phi$ and $\pi^k$ satisfies the second formula $\psi$:

$$\pi \models \phi \mathbf{U} \psi \ \textit{iff} \ \exists i \geq 1, \text{ s.t. } \pi^i \models \psi \text{ and } \forall j < i, \pi^j \models \phi$$

- *Release* (**R**) operator is binary and specifies that for some $i \geq 1$, we have either there exists $j < k$ such that $\pi^j$ satisfies the first formula $\phi$ or $\pi^k$ satisfies the second formula $\psi$:

$$\pi \models \phi \mathbf{R} \psi \ \textit{iff} \text{ either } \exists i > 1, \text{s.t. } \pi^i \models \phi \text{ and } \forall j \leq i, \pi^j \models \psi$$

or

$$\forall k \geq 1, \pi^k \models \psi$$

Figure 9.28 illustrates examples for the semantics of various LTL operators assuming that all examples show on a path $\pi$ in $\boldsymbol{M} = (\boldsymbol{S}, \rightarrow, \boldsymbol{L})$. We can apply LTL to the *Kripke* structure $\boldsymbol{M}_1$ in Figure 9.27 and derive the following formulas:

1. $s_0 \models \mathbf{X}t$ for all path $\pi$, and $s_0 \not\models \mathbf{X}(q \wedge r)$ because the next state of $s_0$ can not satisfy both $q$ and $r$.

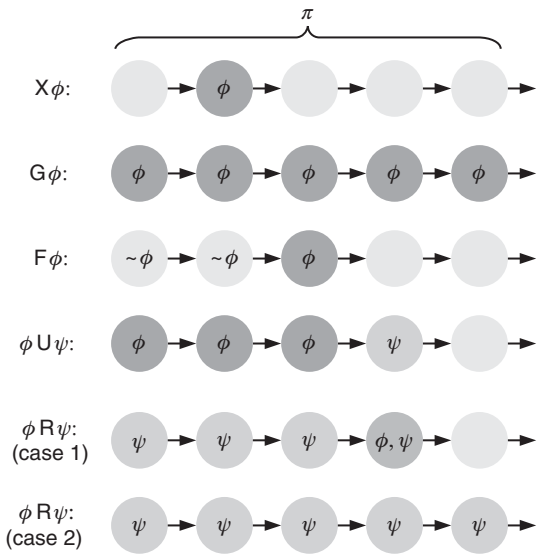



**FIGURE 9.28**

Examples for semantics of LTL operations.

2. $s_0 \models \mathbf{G}\neg(p \wedge t)$ and $s_3 \models \mathbf{G}r$ because $\boldsymbol{M}_1$ can loop at $s_3$ forever.
3. $s_0 \not\models \mathbf{GF}p$ denotes that not every path starting from $s_0$ can finally hold the formula $p$. $\pi = \{s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_3 \ldots\}$ is such one example.
4. $s_0 \models \mathbf{GF}p \rightarrow \mathbf{GF}r$ denotes that every path starting from $s_0$ which satisfies the formula $p$ will always satisfy the formula $r$, but not for the case $s_0 \not\models \mathbf{GF}r \rightarrow \mathbf{GF}p$.
5. $\forall s \in \boldsymbol{S}$ in $\boldsymbol{M}_1$, $s \models \mathbf{X}(q \vee r) \rightarrow \mathbf{F}r$ denotes that the next state of one path starting from every state in $\boldsymbol{M}_1$ can be $q$ or $r$, and then the formula $r$ will also hold on the path finally.

The expressive power of LTL is limited and implicitly quantifies *universally* over paths. An LTL formula can be satisfied if, and only if, all paths starting from the given state satisfy such a formula. A LTL system cannot decide whether one specific formula can be satisfied along some paths in $\boldsymbol{M}$. Therefore, ***computation tree logic*** (CTL), one type of BTTL, is evaluated over a branching-time structure and it quantifies the paths explicitly by introducing both the *existential* operator (**E**) and the *universal* operator (**A**) over paths.

The *Existential* (**E**) operator is defined as follows:

- $\mathbf{EX}\phi$ specifies that there is a path such that $\phi$ holds at the next state:
$$s \models \mathbf{EX}\phi \textit{ iff } \exists\pi = \{v_1 \rightarrow v_2 \rightarrow \ldots v_i \rightarrow \ldots|_{v_1=s}\} \text{ s.t. } v_2 \models \phi$$
- $\mathbf{EG}\phi$ specifies that there is a path along which $\phi$ holds at every state:
$$s \models \mathbf{EG}\phi \textit{ iff } \exists\pi = \{v_1 \rightarrow v_2 \rightarrow \ldots v_i \rightarrow \ldots|_{v_1=s}\} \text{ s.t. } \forall v_i, v_i \models \phi$$
- $\mathbf{EF}\phi$ specifies that there is a path along which $\phi$ holds finally:
$$s \models \mathbf{EF}\phi \textit{ iff } \exists\pi = \{v_1 \rightarrow v_2 \rightarrow \ldots v_i \rightarrow \ldots|_{v_1=s}\} \text{ s.t. } \exists v_i, v_i \models \phi$$
- $\mathbf{E}[\phi\mathbf{U}\psi]$ specifies that there is a path along which $\phi$ holds until $\psi$ holds:
$$s \models \mathbf{E}[\phi\mathbf{U}\psi] \textit{ iff } \exists\pi = \{v_1 \rightarrow v_2 \rightarrow \ldots v_i \rightarrow \ldots|_{v_1=s}\} \text{ s.t. } \pi \models \phi\mathbf{U}\psi$$

The Universal (**A**) operator is defined as follows:

- $\mathbf{AX}\phi$ specifies that for all paths, $\phi$ holds at the next state:
$$s \models \mathbf{AX}\phi \textit{ iff } \forall\pi = \{v_1 \rightarrow v_2 \rightarrow \ldots v_i \rightarrow \ldots|_{v_1=s}\} \text{ s.t. } v_2 \models \phi$$
- $\mathbf{AG}\phi$ specifies that for all paths, $\phi$ holds at every state of the path:
$$s \models \mathbf{AG}\phi \textit{ iff } \forall\pi = \{v_1 \rightarrow v_2 \rightarrow \ldots v_i \rightarrow \ldots|_{v_1=s}\} \text{ s.t. } \forall v_i, v_i \models \phi$$
- $\mathbf{AF}\phi$ specifies that for all paths, $\phi$ holds finally:
$$s \models \mathbf{AF}\phi \textit{ iff } \forall\pi = \{v_1 \rightarrow v_2 \rightarrow \ldots v_i \rightarrow \ldots|_{v_1=s}\} \text{ s.t. } \exists v_i, v_i \models \phi$$

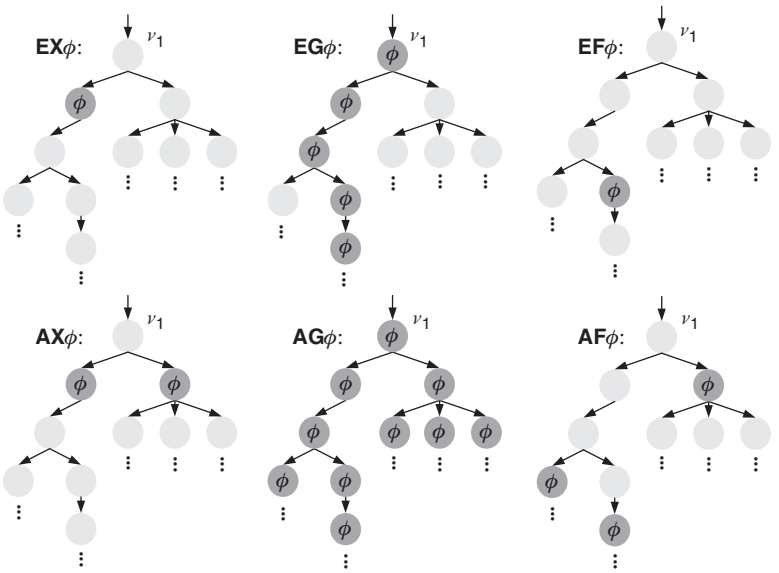



**FIGURE 9.29**

Illustrations for CTL *Existential* and *Universal* operations.

- $\mathbf{A}[\phi \mathbf{U} \psi]$ specifies that for all paths, $\phi$ holds until $\psi$ holds:

$$s \models \mathbf{A}[\phi \mathbf{U} \psi] \textit{iff } \forall \pi = \{v_1 \rightarrow v_2 \rightarrow \ldots v_i \rightarrow \ldots |_{v_1 = s}\} \text{ s.t. } \pi \models \phi \mathbf{U} \psi$$

Figure 9.29 illustrates partial examples for the *Existential* and *Universal* operations according to the preceding definitions.

CTL is capable of specifying branching behaviors such as **AG(EF***f*), which is also known as **resetability—**meaning there is always a path back to *f*. This property cannot be modeled by LTL because of the lack of the path quantifier **E**. Likewise, there exists some LTL formulas that cannot be expressed in CTL. For example, $\mathbf{FG}\phi$ in LTL means that the formula $\phi$ will finally hold along every path from the given point. Its semantic should be expressible as $\mathbf{A}(\mathbf{FG}\phi)$. However, in CTL, every temporal operator (**F** and **G**) must be preceded by a path quantifier (**E** or **A**). Hence, CTL cannot express $\mathbf{A}(\mathbf{FG}\phi)$. CTL* extends the expressiveness from both LTL and CTL and primarily allows a path quantifier to be used followed by an arbitrary LTL formula. The relationships between the expressiveness of LTL, CTL, and CTL* can be viewed as LTL $\cup$ CTL $\subset$ CTL*, which are illustrated in Figure 9.30. Particularly, there is a set $\{\phi_4\}$ of CTL* formulas that can be expressed neither in CTL nor in LTL. $\mathbf{E}(\mathbf{GF}\phi)$ is such an example, saying that there is a path where from one certain state, $\phi$'s holds through arbitrarily many states to the end [Huth 2004].

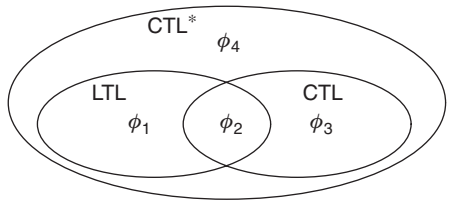



**FIGURE 9.30**

Relationships between the expressiveness of LTL, CTL, and CTL*.

The properties of design systems can be divided into two types [Owicki 1982]:

1. **Safety properties** that indicate that some bad event will never happen. For a sequential program, safety guarantees that no incorrect outcome will be produced by the program. For a finite state machine, safety checking denotes those properties whose violation can always find a finite trace. Another typical example of safety is a mutual exclusive property that states that having more than one process in the critical section will never occur.
2. **Liveness properties** that indicate that some good event will eventually happen. For a sequential program, the program will terminate as it produces a legal outcome. For a finite state machine, those properties that may be violated will never have a finite witness. CTL can model the simple liveness for the phrase "The light will turn green" as $light \models \mathbf{AF}(green)$. "Any request will eventually be satisfied" is another example semantic phrase that can be expressed and the corresponding CTL expression is $\mathbf{AG}(Req) \Rightarrow \mathbf{AF}(Sat)$. Liveness focuses on a slice in the tree structure and may incur the witness as a computation path of infinite steps.

To illustrate the safety and liveness properties, consider a two-input Muller C-element used for asynchronous circuit connections. Figure 9.31a shows its gate-level netlist with two Boolean inputs ($x$, $y$) and one output ($z$). The corresponding dynamic behavior is represented by the state transition graph in Figure 9.31b.

A safety property of the C-element is that if all inputs and outputs are equal, then the output $z$ will not change its value until all inputs flip their values. There are two situations: all values are 0 and all values are 1.

- $\mathbf{AG}((x = 0 \wedge y = 0 \wedge z = 0) \Rightarrow \mathbf{A}(z = 0\ \mathbf{U}\ (x = 1 \wedge y = 1)))$
- $\mathbf{AG}((x = 1 \wedge y = 1 \wedge z = 1) \Rightarrow \mathbf{A}(z = 1\ \mathbf{U}\ (x = 0 \wedge y = 0)))$

A liveness property of the C-element is that if both inputs become equal, then the output $z$ will eventually change to the corresponding value. There are two situations: both input values are 0 and both input values are 1.

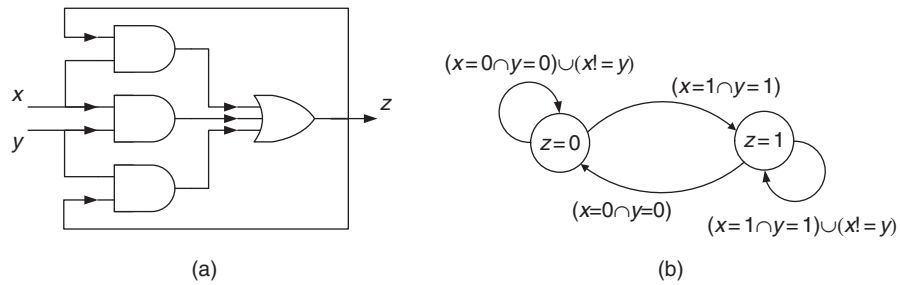



**FIGURE 9.31**

(a) Gate-level netlist. (b) state transition graph of a C-element.

- **AG(A**$(x = 0 \wedge y = 0)$ **U** $(z = 0 \vee x = 1 \vee y = 1)$)
- **AG(A**$(x = 1 \wedge y = 1)$ **U** $(z = 1 \vee x = 0 \vee y = 0)$)

### 9.5.2.1 *Model checking with temporal logic*

Let a *Kripke* structure $\boldsymbol{M} = (\boldsymbol{S}, \rightarrow, \boldsymbol{L})$ represent a finite state concurrent system. The model-checking problem can be formulated as: given a model $\boldsymbol{M}$, a property $p$ specified as a temporal formula, and a state $s$, does $s \models p$ hold in $\boldsymbol{M}$? The corresponding result is either (1) yes, $s \models p$ in $\boldsymbol{M}$, or (2) no, $s \not\models p$ in $\boldsymbol{M}$. Especially for the latter case, such a result is derived from finding a counterexample that invalidates $p$ in $\boldsymbol{M}$. Therefore, the modeling checking problem can be addressed by computing the state set $\boldsymbol{S}_p$ that satisfies $p$ in $\boldsymbol{M}$.

The **labeling algorithm,** proposed by E. Clarke, E. Emerson, and A. Sistla [Clarke 1986], is a basic algorithm for the model checking problem. Given a CTL formula, the labeling algorithm labels the set of states in which the target formula $p$ holds, which is denoted as $[[p]] \triangleq \{\forall s \in \boldsymbol{S} \text{ in } \boldsymbol{M}, s \models p\}$, and called the **denotation** of $p$. Deriving $[[p]]$ starts by decomposing $p$ into a set of subformulas in a *bottom up* manner. Because $\{\perp, \neg, \wedge\}$ and {**AF**, **EX**, **EU**} can form an adequate set of connectives for CTL [Martin 2004], and all other propositional and temporal connectives can be written in terms of this set, a preprocessing step to convert the target formula $p$ into an equivalent form in terms of this adequate set is first invoked and then followed by labeling states in $\boldsymbol{M}$ for $[[p]]$. Later, the denotation $[[p]]$ is compared with the set $\boldsymbol{S}_{init}$ of all initial states to check whether $\boldsymbol{S}_{init} \subseteq [[p]]$.

The labeling algorithm explicitly enumerates the states in the model whose size often grows exponentially in terms of the numbers of variables in the system. This problem is typically referred to as the *state explosion problem*. To overcome this issue, a more efficient technique called **fix-point computation** is proposed, which incorporates OBDD for symbolic computation and implicit representation of states. Model checking with OBDDs is often referred to as **symbolic model checking** [Burch 1990], and **SMV,** developed at Carnegie Mellon University, is one such verifier [McMillan 1992].

*Fix-point computation* finds the set of states that satisfies the specific *global* CTL formula. A function $x_{i+1} = f(x_i)$ is called a **fix-point** if $\exists x_k$, where $k \geq 0$, s.t. $x_{k+1} = f(x_k) = x_k$. Given a starting value $x_0$, a **fix-point** can be found by iteratively mapping $f$ to $x_i$ until $f(x_k) = x_k$. To help calculate the fix-points on a *Kripke* structure $\boldsymbol{M} = (\boldsymbol{S}, \rightarrow, \boldsymbol{L})$, we define a function $\tau$ called a *predicate transformer*, which takes a subset of $\boldsymbol{S}$ and outputs another subset. In other words, the function $\tau$ is defined on the basis of the power set $P(\boldsymbol{S})$, which is the set of all subsets of $\boldsymbol{S}$. $\tau^i(\boldsymbol{S}')$ denotes $i$ applications of $\tau$ to the given subset $\boldsymbol{S}' \subseteq \boldsymbol{S}$. That is,

$$\tau^i(\boldsymbol{S}') = \underbrace{\tau(\tau(\ldots(\tau}_{i \text{ times}}(\boldsymbol{S}'))\ldots)$$

$\tau$ is *monotonic*, provided that for any two subsets of $\boldsymbol{S}$, $\boldsymbol{P}$, and $\boldsymbol{Q}$, if $\boldsymbol{P} \subseteq \boldsymbol{Q} \subseteq P(\boldsymbol{S})$, then $\tau(\boldsymbol{P}) \subseteq \tau(\boldsymbol{Q})$. Note that because $\tau$ is monotonic, by starting from a subset of $\boldsymbol{S}$ and continuously applying $\tau$, a fixed point can always be reached.

Let $\tau$ be monotonic, $\emptyset$ be the empty set, and $\mathbf{U}$ be a finite set $\{s_0, s_1, \ldots, s_n\} \subseteq P(\boldsymbol{S})$ of $n$ elements in $\boldsymbol{M}$, then $\exists l$, s.t. $\tau^l(\emptyset) = \tau^{l+1}(\emptyset)$ and $\exists u$, s.t. $\tau^u(\mathbf{U}) = \tau^{u+1}(\mathbf{U})$. $\tau^l(\emptyset)$ and $\tau^u(\mathbf{U})$ are called the *least* and *greatest* fix-points of $\tau$, which are denoted by $fp_{min}$ and $fp_{max}$, respectively. Each basic CTL* operator can be further represented by either $fp_{min}$ or $fp_{max}$ over an appropriate predicate transformer. For a complete treatment of the underlying theory and proof, please refer to [Granas 2003].

Suppose that we would like to apply the fix-point computation to check $\mathbf{AG}(\phi \Rightarrow \mathbf{AF}\psi)$, then sub-annotations will be computed in a bottom-up manner. That is then $[[\psi]]$, $[[\mathbf{AF}\psi]]$, $[[\phi]]$, $[[\phi \Rightarrow \mathbf{AF}\psi]]$, and $[[\mathbf{AG}(\phi \Rightarrow \mathbf{AF}\psi)]]$ in this example. Assuming $\psi = p$ and $\phi = q$, let's check the process of calculating the formula on the basis of the example given in Figure 9.27.

- $[[\psi]] = [[r]] = \{s_3\}$
- $[[\mathbf{AF}\psi]] = [[\mathbf{AF}r]] = \{s_0, s_1, s_2, s_3\}$ can be computed as the union of
  - $[[\psi]] = [[r]] = \{s_3\}$
  - $[[r \vee \mathbf{AX}r]] = \{s_3\} \mathrm{U} \{s_1, s_2\} = \{s_1, s_2, s_3\}$
  - $[[r \vee \mathbf{AX}(r \vee \mathbf{AX}r)]] = \{s_3\} \mathrm{U} \{s_0, s_1, s_2, s_3\} = \{s_0, s_1, s_2, s_3\}$
  - no need to repeat since $\{s_0, s_1, s_2, s_3\}$ converges
- $[[\phi]] = [[p]] = \{s_0\}$
- $[[\phi \Rightarrow \mathbf{AF}\psi]] = [[\neg\phi \vee (\mathbf{AF}\psi)]] = [[\neg p \vee (\mathbf{AF}r)]]$
  - $[[\neg p \vee (\mathbf{AF}r)]] = \{s_1, s_2, s_3\} \mathrm{U} \{s_0, s_1, s_2, s_3\} = \{s_0, s_1, s_2, s_3\}$
- $[[\mathbf{AG}\mu]]$ can be computed as the intersection of $[[\mu]]$, $[[\mu \wedge \mathbf{AX}\mu]]$, $[[\mu \wedge \mathbf{AX}(\mu \wedge \mathbf{AX}\mu)]]$, and etc. Therefore, $[[\mathbf{AG}(\phi \Rightarrow \mathbf{AF}\psi)]]$ can be obtained from the following and result in $\{s_0, s_1, s_2, s_3\}$:
  - $[[\mu]] = [[\phi \Rightarrow \mathbf{AF}\psi]] = \{s_0, s_1, s_2, s_3\}$
  - $[[\mu \wedge \mathbf{AX}\mu]] = \{s_0, s_1, s_2, s_3\} \cap \{s_0, s_1, s_2, s_3\} = \{s_0, s_1, s_2, s_3\}$
  - $[[\mu \wedge \mathbf{AX}(\mu \wedge \mathbf{AX}\mu)]] = \{s_0, s_1, s_2, s_3\}$
  - ... all remaining computations converge to $\{s_0, s_1, s_2, s_3\}$

Because every state belongs to $[[\mathbf{AG}(\phi \Rightarrow \mathbf{AF}\psi)]] = \{s_0, s_1, s_2, s_3\}$, the *Kripke* structure $\boldsymbol{M} = (\boldsymbol{S}, \rightarrow, \boldsymbol{L})$ satisfies this property. As we can see, computing the state set for propositional connectives is straightforward. The computation for temporal connectives such as $\mathbf{EX}\phi$ is relatively sophisticated and requires applying the temporal operations over the current state set repeatedly until there is no change.

Symbolic model checking is often limited by the sizes of corresponding OBDDs used in the computation. Typically, a good variable ordering is crucial for minimizing OBDD size. However, finding optimal ordering is a proven NP-complete problem. In some cases, even with the best ordering, the OBDD size is still larger than the available computation resource. To address this problem, an alternative method, called ***bounded model checking*** (BMC), was proposed, which only tries to find counterexamples for properties within a bounded number of clock cycles (state transitions). Most of the bounded model checkers use a propositional decision (SAT) procedure [Biere 1999]. Several efficient satisfiability solvers have been developed in recent years that are capable of solving problems with more than thousands of variables. Bounded model checking can find minimal length counterexamples as the propositional decision procedure traverses the state-transition graph step by step. This feature can also make users easily understand counterexamples and consequently facilitate the debugging process.

Given the *Kripke* structure $\boldsymbol{M} = (\boldsymbol{S}, \rightarrow, \boldsymbol{L})$ and a safety property $\phi$, by use of BMC we can determine whether a length-$k$ execution path of $\boldsymbol{M}$ that satisfies $\phi$ exists. That is, $\boldsymbol{M} \models_k \mathbf{E}\phi$. Let a propositional formula $T(s,s')$ define the relationship of the state transition in $\boldsymbol{M}$ and let $\boldsymbol{I}(s)$, a predicate over the state variables, define the initial states. The BMC problem is equivalent to the satisfiability problem of a Boolean formula $[[\boldsymbol{M}, \phi]]_k = [[\boldsymbol{M}]]_k \wedge [[\phi]]_k$ where $[[\boldsymbol{M}]]_k$ and $[[\phi]]_k$, respectively, encode the set of length-$k$ execution paths of $\boldsymbol{M}$ and the set of length-$k$ paths that satisfy $\phi$ in $\boldsymbol{M}$.

For a valid length-$k$ path $\pi = \{s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k\}$, $[[\boldsymbol{M}]]_k$ can be defined as

$$[[\boldsymbol{M}]]_k = I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) = I(s_0) \wedge \prod_{i=0}^{k-1} T(s_i, s_{i+1})$$

The core of encoding for a formula $\phi$ with $k$ steps depends on whether $\boldsymbol{M}$ contains any loop that starts at $s_l$ and ends at $s_k$. Therefore, $[[\phi]]_k$ can be computed as the disjunction of two cases:

1. Without loopback in $\boldsymbol{M}$: $[[\phi]]_k \triangleq \left(\neg\left(\prod_{l=0}^{k} T(s_l, s_k) \wedge [[\phi]]_k^0\right)\right)$, where for every $[[.]]_k^i$, $k$ is the length of the prefix of the path and $i$ is the current position in this prefix.
2. With a loopback in $\boldsymbol{M}$: $[[\phi]]_k \triangleq \prod_{l=0}^{k} (T(s_l, s_k) \wedge {}_l[[\phi]]_k^0))$, where for every ${}_l[[.]]_k^i$, $i$ is the current position in the path $\pi$, $k$ is the length of the prefix of this path, and $l$ is the position where the loop starts.

For example, given a formula $\phi = \mathbf{F}p$, $\boldsymbol{M} \models_k \phi$ is used to check whether any reachable state in which a property $p$ holds in $\boldsymbol{M}$ within $k$ steps exists. Bounded model checking will first derive $[[\boldsymbol{M}, \phi]]_k = I(s_0) \wedge \prod_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \prod_{j=0}^{k} p(s_j)$, where $p(s_j) = 1$ if the property $p$ holds on $s_j$, otherwise $p(s_j) = 0$. This satisfiability problem can be solved with an SAT solver. It will return 1 if such a path is found. To check whether any reachable state that satisfies $p$, provided that $q$ holds infinitely (*i.e.,* $\phi = \mathbf{GF}q \wedge \mathbf{F}p$) exists, modeling the loopback behavior in $\boldsymbol{M}$ is required. That is,

$$[[\boldsymbol{M}, \phi]]_k = I(s_0) \wedge \prod_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \prod_{j=0}^{k} p(s_j) \wedge \prod_{l=0}^{k} \left(T(s_l, s_k) \wedge_l [[q]]_k^0\right)$$

Although bounded model checking with the propositional decision (**SAT**) procedure can handle larger circuits, it is an incomplete technique. If the checking formulas are unsatisfiable (*i.e.,* the property holds true over a bounded length $k$ of checking, there is no guarantee that the property will hold or not over a length greater than $k$.

### 9.5.3 Theorem proving

We have introduced how propositional and temporal logic can be automated to compare two representations in equivalence checking and to validate properties from the specifications against a given model in model checking. The effectiveness of both equivalence checking and model-checking techniques is often limited by the capacity and performance of the underlying engines used such as OBDD and SAT. Sometimes, the complexity of a verification task for an arithmetic circuit, such as a data path or a signal processing unit, can be reduced if a more general mathematical formulation of the circuit, with a better abstraction of the word-level information, is provided. Theorem proving techniques are applied for such purposes.

Theorem proving is the process for determining whether a given implementation satisfies the target specification by means of mathematical reasoning, as shown in Figure 9.32. Both the implementation and specification need to be transformed into formulas in a formal logic system. The relationships between implementation and specification are regarded as theorems in logic. The conformance is then established by proving the theorems either from implementation

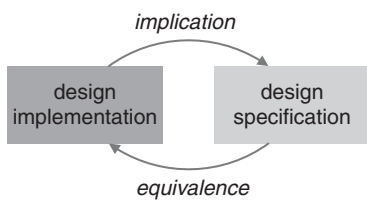



**FIGURE 9.32**

Verification by theorem proving.

to specification, denoted by the *implication* arrow in the figure, or from specification to implementation, denoted by the *equivalence* arrow.

A *proof system* (or *calculus*) ***S*** consists of:

1. *Expressions* of ***S***: a finite sequence of symbols
2. *Well-formed formulas* of ***S***: a subset of the expressions of ***S***
3. *Axioms* of ***S***: a finite set of the well-formed formulas of ***S***
4. *Inference rules* of ***S***: a finite set of derivation rules from a given finite set of well-formed formulas to a new well-formed formula

The general form of an inference rule is $\frac{\alpha_1, \alpha_2, \ldots \alpha_k}{\beta}$, where the well-formed formulas $\alpha_1, \alpha_2, \ldots, \alpha_k$ are called the *premises* of the rule, whereas the well-formed formula $\beta$ is called the *conclusion*.

In such a proof system ***S***, a *proof* is a finite sequence of formulas, $\phi_1, \phi_2, \ldots, \phi_n$ in which $\phi_i$ can be either an *axiom* or else *derived* from applying an inference rule of ***S*** over $\{\phi_1, \phi_2, \ldots, \phi_{i-1}\}$, which is denoted as $\{\phi_1, \phi_2, \ldots, \phi_{i-1}\} \vdash \phi_i$. The last formula $\phi_n$ is the goal of the proof, which is known as a *theorem* of ***S***. Sometimes, proofs may require supplementary *assumptions*, such as $\Gamma = \{\psi_1, \psi_2, \ldots, \psi_{i-1}\}$ from the domain specific axioms. The term $\Gamma \vdash \phi$ asserts that the formula $\phi$ is valid if all assumptions in $\Gamma$ are true. If $\Gamma$ is empty, we write this as $\vdash \phi$.

Many modern theorem proving systems are publicly available. These include Coq [Coq 2003], *Z/Eves* [Saaltink 1999], ***High-Order Logic*** (HOL) [Nipkow 2002], *PVS* [Owre 1992], and *ACL2* [Kaufmann 2002]. To illustrate the deduction process involved in theorem proving, we use HOL, developed at the University of Cambridge [Gordon 1993], for the remainder of the discussion. HOL supports the use of standard predicate operators, five axioms, and eight primitive inference rules, which are listed in Table 9.3, for expressing most ordinary mathematical theories.

The first step of the proof method in HOL is to formalize both the specification and the implementation into the formal logic used in the proof system. Then, the formulation of a proof goal can be achieved by either proof of implication (forward) or proof of equivalence (backward) with the inference rules. In the forward manner, a theorem prover starts with simple lemmas that can be proven directly to develop new rules. Rules are successively combined into more difficult lemmas until the target theorem is proven. Figure 9.33 shows an example for such an HOL theorem proving. The functional specification of the underlying black-box, shown in Figure 9.33a, is an NOR function denoted by $f = \bar{x} \times \bar{y}$. Its formal specification can be expressed as $SPEC(x, y, z) \triangleq z = (\neg x \wedge \neg y)$. And the implementation, which is shown in Figure 9.33b, may use only primitive AND, OR, and NOT gates. The corresponding descriptions of these gates in formal logic are:

- $\text{AND}(i_1, i_2, out) \triangleq out = (i_1 \wedge i_2)$, where $i_1$ and $i_2$ are input ports and *out* is an output port.

**Table 9.3** Base Rules of Higher Order Logics Used in HOL

| Name | Explanation | Rule | Remark |
|---|---|---|---|
| ASSUME | Assumption introduction | $\frac{-}{t \vdash t}$ | |
| REFL | Reflexivity | $\frac{-}{\vdash t = t}$ | |
| ABS | Abstraction | $\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.t_1) = (\lambda x.t_2)}$ | If $x$ is not free in $\Gamma$, where $(\lambda x.t_i)$ denotes the function defined by $f(x) = t_i$ |
| BETA_CONV | Beta-conversion | $\frac{-}{\vdash (\lambda x.t_1)t_2 = t_1[t_2/x]}$ | $t_1[t_2/x]$ substitutes $t_2$ for $x$ in $t_1$ with the restriction that no free variables in $t_2$ become bound after substitution into $t_1$ |
| SUBST | Substitution | $\frac{\Gamma_1 \vdash t_1 = t_2 \mid \Gamma_2 \vdash t[t_1]}{\Gamma_1 \cup \Gamma_2 \vdash t[t_2]}$ | $t[t_i]$ denotes a term $t$ containing a subterm $t_i$ |
| INST_TYPE | Type instantiation | $\frac{\Gamma \vdash t}{\Gamma \vdash t[\sigma_1, \ldots, \sigma_n/v_1, \ldots, v_n]}$ | $t[\sigma_1, \ldots, \sigma_n/v_1, \ldots, v_n]$ substitutes in parallel the types $\sigma_1, \ldots, \sigma_n$ for the variables $v_1, \ldots, v_n$ in $t$ |
| DISCH | Assumption discharging | $\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \Rightarrow t_2}$ | $\Gamma - \{t_1\}$ denotes the set subtracting $\{t_1\}$ from $\Gamma$ |
| MP | Modus ponens | $\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \mid \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \mid - t_2}$ | |

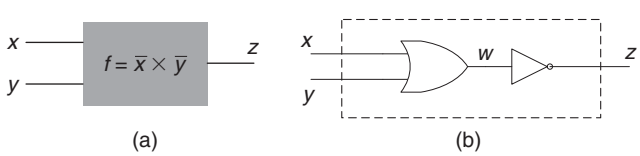



**FIGURE 9.33**

Example of theorem proving by HOL.

- $\mathrm{OR}(i_1, i_2, out) \triangleq out = (i_1 \vee i_2)$, where $i_1$ and $i_2$ are input ports and *out* is an output port.
- $\mathrm{NOT}(i, out) \triangleq out = (\neg i)$, where $i$ is an input port and *out* is an output port.

Therefore, the formal definition for the implementation in Figure 9.33b is $IMPL(x, y, z) \triangleq \exists w.OR(x, y, w) \wedge NOT(z, w)$. The goal of this proof is to derive *SPEC*(*x, y, z*) from *IMPL*(*x, y, z*) by applying the inference rules specified in Table 9.3. The proof — given step-by-step — is as follows in Table 9.4. Please note that the actual process executed with HOL software may not look exactly the same though. However, it should be similar to what it is shown below.

**Table 9.4** Step-by-step Proof for an NOR Function

| **Proof** | |
|---|---|
| *IMPL*(*x, y, z*) | {from the circuit diagram} |
| $\vdash \exists w.OR(x, y, w) \wedge NOT(z, w)$ | {by definition of the implementation} |
| $\vdash OR(x, y, w) \wedge NOT(z, w)$ | {strip off $\exists w$} |
| $\vdash (w = x \vee y) \wedge NOT(z, w)$ | {by formal definition of OR gate} |
| $\vdash (w = x \vee y) \wedge (z = \neg w)$ | {by formal definition of NOT gate} |
| $\vdash (z = \neg(x \vee y))$ | {substitute $w$ with $x \vee y$} |
| $\vdash (z = \neg x \wedge \neg y)$ | {distribute $\neg$ over $x \vee y$ } |
| $\vdash SPEC(x, y, z)$ | {by definition of the specification} |
| $\vdash IMPL(x, y, z) \Rightarrow SPEC(x, y, z)$ | |
| **Q.E.D.** | |

Theorem proving can be applied to verify implementations described at different levels of abstraction. The formal specification of the behavior of a transistor-level CMOS inverter, for example, can be expressed by $SPEC(x, y) \triangleq y = (\neg x)$ [Gordon 1992]. Consider the network structure shown in Figure 9.34. The implementation is built on basic modules and includes a power cell, a ground cell, a P-type transistor, and an N-type transistor which are denoted as *VDD*(*p*), *GND*(*q*), *PTran*(*x*, *p*, *y*), and *NTran*(*x*, *y*, *q*), respectively. The behaviors of these basic modules can be formally defined as:

- $VDD(p) \triangleq (p = \top(\text{true}))$
- $GND(q) \triangleq (q = \bot(\text{flase}))$
- $PTran(x, p, y) \triangleq (\neg x \Rightarrow (p = y))$
- $NTran(x, y, q) \triangleq (x \Rightarrow (y = q))$

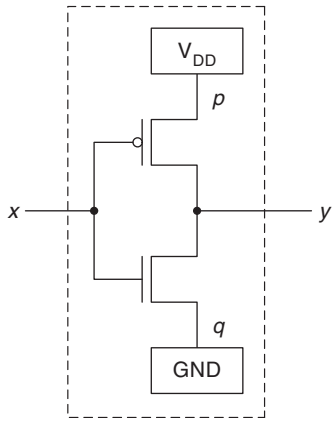



**FIGURE 9.34**

CMOS inverter.

Then, the entire network structure can be formulated as:

$$IMPL(x,y) \triangleq \exists p,q.V_{DD}(p) \wedge PTran(x,p,y) \wedge Ntran(x,y,q) \wedge GND(q)$$

Again, the proof goal is to derive $SPEC(x, y)$ from $IMPL(x, y)$ by applying inference rules. The step-by-step proof process is as follows.

**Proof**

$IMPL(x,y)$ {from the network structure }

$\vdash \exists p,q.VDD(p) \wedge PTran(x,p,y) \wedge NTran(x,y,q) \wedge GND(q)$
{by definition of the implementation }

$\vdash VDD(p) \wedge PTran(x,p,y) \wedge NTran(x,y,q) \wedge GND(q)$
{strip off $\exists p,q$ }

$\vdash (p = \top) \wedge PTran(x,p,y) \wedge NTran(x,y,q) \wedge (q = \bot)$
{by definition of *VDD* and *GND* cells }

$\vdash (p = \top) \wedge PTran(x,\top,y) \wedge NTran(x,y,\bot) \wedge (q = \bot)$
{substitute $p$ in *PTran*, $q$ in *NTran* }

$\vdash (\exists p.p = \top) \wedge PTran(x,\top,y) \wedge NTran(x,y,\bot) \wedge (\exists q.q = \bot)$
{use $\exists a.t_1 \wedge t_2 = (\exists a.t_1) \wedge t_2$ if $a$ is free in $t_2$ }

$\vdash (\top) \wedge PTran(x,\top,y) \wedge NTran(x,y,\bot) \wedge (\top)$
{use$(\exists a.a = \top) = \top$ and $(\exists a.a = \bot) = \top$ }

$\vdash PTran(x,\top,y) \wedge NTran(x,y,\bot)$
{use $(x \wedge \top) = x$ }

$\vdash (\neg x \Rightarrow (\top = y)) \wedge (x \Rightarrow (y = \bot))$
{by definition of *PTran* and *NTran* cells }

$\vdash (x \vee (\top = y)) \wedge (\neg x \vee (y = \bot))$
{by$(a \Rightarrow b) = (\neg a \vee b)$ }

$\vdash (x \wedge \neg x) \vee (x \wedge (y = \bot)) \vee ((\top = y) \wedge \neg x) \vee ((\top = y) \wedge (y = \bot))$

$\vdash (\bot) \vee (x \wedge (y = \bot)) \vee ((\top = y) \wedge \neg x)(\bot)$

$\vdash (x \wedge (y = \bot)) \vee ((\top = y) \wedge \neg x)$

{apply Boolean simplification }

$\vdash y = (\neg x)$

{if $x = \top \Rightarrow (y = \bot)$ and if $x = \bot \Rightarrow (y = \top)$ }

$\vdash IMPL(x, y) \Rightarrow SPEC(x, y)$

**Q.E.D**.

Theorem proving has been successfully applied to the verification of hardware designs, such as the TAMARACK microprocessor [Joyce 1986] and the Viper microprocessor [Cohn 1988]. Its strength is its ability to support the expressiveness of higher order logics, to relate circuit behaviors at different levels of abstraction [Melham 1988], and to provide many effective reasoning utilities. Moreover, the design hierarchy and regularity can be exploited by theorem provers, which enable users to be in full control of the verification process. Higher order logics can specify and verify generic and parameterized hardware designs. One such example would be a channel encoder with words in $n$-bit width. Also, tactics of inference rules can continuously evolve during the deduction process. Particularly frequent and useful theories/theorems can be customized and retained for future proofs.

Verification by theorem proving requires users to familiarize themselves with the proof system and to spend a considerable amount of effort toward developing the formal models for both the specification and the implementation. This is one of the major disadvantages of the approach. Moreover, because of the lack of sound proof systems for higher order logic, the derivation of inference rules may require a great deal of human intervention, especially for complex and large theorems. For these reasons, the application of theorem proving has been limited and not widely used for industrial design projects.

## 9.6 ADVANCED RESEARCH

Simulation remains the mainstream verification approach in the industry. Its scalability, along with its easy applicability to designs at almost any abstraction level, makes it attractive for complex verification tasks. When used as a standalone technique, simulation can detect simple and easy-to-find bugs. Its effectiveness in finding corner-case, hard-to-detect bugs can be limited because of the availability of high-quality stimuli that can cover a wide range of the corner cases and can activate and reveal the subtle bugs. Although traditional formal techniques—broadly speaking, model checking and theorem proving—can, in principle, analyze and find subtle bugs, their applicability can be limited by their runtime inefficiency and/or difficulty in use.

For simulation-based approaches, measuring the coverage and preparing the test vectors are the two most important things in the verification plan. The coverage-driven verification (CDV) flow, shown in Figure 9.6, links these two together and can be automated if the test generation constraints can be modified automatically [Bai 2003; Chen 2003; Wen 2006, 2007]. Such improvements can substantially save the amount of manual efforts needed for coverage analysis and test preparation. The improvements in coverage-driven verification can be divided into two categories: feedback-based coverage-driven verification and coverage-driven verification by construction.

**Feedback-based coverage-driven verification** modifies the biases and seeds to direct the automatic test generation. A generic algorithm [Bose 2001] can be applied to resynthesize test cases for optimizing the coverage. The authors in [Tasiran 2001] represent the DUV as a Markov chain model and analyze the feedback data to modify the model's parameters. The authors in [Fine 2003] cast the coverage-driven test generation in a statistical inference framework by modeling the relationship between coverage information and the directives to the test generation as Bayesian networks. A machine-learning–based technique in [Fine 2006] was later proposed to provide enhanced coverage through automatically learning the relationship between the initial state and vector generation success.

**Coverage-driven verification by construction** derives an abstract model that can capture the logical constraints in the DUV and assemble the new directives to correctly hit the uncovered events. [Ur 1999] abstract the processor control as a set of FSMs and use them to automate the verification tasks. A physical test case is derived from a sequential trace of the state traversal in the FSM. The works in [Chen 2003] and [Bai 2003] generate tests to target stuck-at and crosstalk faults in processors and use a ***virtual constraint circuit*** (VCC) for assisting the module-level test generation process. The application is for ***software-based self-test*** (SBST) [Lai 2000]. A data-mining approach based on simulation data was proposed in [Wen 2006, 2007] to approximate the functionality of the DUV as BDDs that can then be used to better guide the test generation process.

Although the capacity and performance of formal methods has improved significantly over the past decade, such improvements barely kept pace with the growth in design complexity. The search for new solutions resulted in some powerful hybrid techniques that combined formal and informal approaches. These hybrid techniques attempt to address verification bottlenecks by enhancing coverage of the state space traversed.

Researchers who investigated formal methods have widely recognized the importance of providing a way to combine disparate tools. Joyce and Seger experimented with combining *trajectory evaluation* with theorem proving. They used trajectory evaluation as a decision procedure for the ***higher-order***

***logic*** (HOL) system [Joyce 1993]. A proposal called interface logics [Guttman 1991] discusses the idea of combining different *theorem provers* by defining a single logic such that the logic of each individual tool can be viewed as its sublogics. [Jang 1997] used CTL model checking to verify a set of properties of embedded microcontrollers, and the proof of the top-level specification was achieved through a compositional argument by use of the properties instead of through a theorem prover. A hybrid of two model-checking techniques, called MIST [Hazelhurst 2002], enables a handshake between *symbolic trajectory evaluation* and *symbolic model checking*.

Generally speaking, **hybrid methods** combining formal and informal techniques aim to increase the design space coverage and, thus, the probability of finding design errors. These types of methods include *control space exploration*, *directed functional test generation*, *combining ATPG with formal techniques*, and *heuristic-based traversal*. Control space exploration addresses the problem of finding bugs and increases space coverage by exploring control logic [Iwashita 1994; Ho 1995; Geist 1996; Moondanos 1998]. Directed functional test generation leverages the strengths of both formal verification and simulation techniques to generate functional tests [Sumners 2000; Ganai 2001; Mishra 2005]. Because ATPG can avoid state space explosion by use of dual justification and propagation techniques to localize the search, adding formal techniques can compensate for the inherent incompleteness of ATPG, making the combination a more complete and effective verification approach [Boppana 1999; Huang 2001; Vedula 2004]. Heuristic-based traversal tackles the need to efficiently traverse state space by an extensive use of heuristics [Yang 1998; Wagner 2005; Shyam 2006]. Note that because of the inherent incompleteness of informal techniques, any method that combines an informal technique with another is also an incomplete verification method.

## 9.7 CONCLUDING REMARKS

This chapter reviews the basic concepts of functional verification and the challenges associated with it. Different levels of the verification hierarchy, including the designer level, unit level, core level, chip level, and system/board level, are explained. Various coverage metrics used for measuring the explored extent of verification are provided. The simulation-based approach is currently the most pervasive form of verification. Key components such as testbench and simulation environment development are reviewed. The emerging assertion-based verification method is explained in detail. To compensate for the incompleteness of simulation-based verification, formal methods built on mathematical theories were developed. Basic concepts in equivalence checking, model checking, and theorem proving are reviewed. Current research efforts toward advancing functional verification are summarized to conclude this chapter.

## 9.8 EXERCISES

**9.1.** (**Line Coverage**) Suppose that the module in Box 9.11 was specified in your Verilog HDL design:

**BOX 9.11**

```
 1. module test;
 2. reg X, Y, Z;
 3. initial
 4. begin
 5.    X = 1'b0;
 6.    Y = 1'b1;
 7. if (X)
 8.    Z = Y;
 9. else
10.    Z = ~Y;
11. end
12. endmodule
```

Calculate the line coverage after simulation and identify the line or lines that has/have not been covered.

**9.2.** (**Toggle Coverage**) Suppose that the following module in Box 9.12 was specified in your Verilog HDL design:

**BOX 9.12**

```
 1. module test;
 2. reg [2:0] X;
 3. initial
 4. begin
 5.    X = 3'b000;
 6.    #100;
 7.    X = 3'b110;
 8.    #100;
 9.    X = 3'b010;
10.    #100;
11. end
12. endmodule
```

After simulation, the register would have achieved a total toggle percentage of 50%. Please identify which toggles are missing.

**9.3.** (**Expression Coverage**) Suppose that the following module was specified in your Verilog HDL design:

**BOX 9.13**

```
 1. module test;
 2. reg X, Y;
 3. wire Z;
 4. assign Z = X|Y;
 5. initial
 6. begin
 7.    X = 1'b0;
 8.    Y = 1'b0;
 9.    #50;
10.    X = 1'b1;
11.    #50;
12.    Y = 1'b1;
13.    #50;
14. end
15. endmodule
```

This module consists of only one expression: X|Y. Calculate the expression coverage after simulation and identify those cases that are not covered.

**9.4.** (**FSM Coverage**) Suppose that the module in Box 9.14 was specified in your Verilog HDL design:

**BOX 9.14**

```
 1. module test;
 2. reg [1:0] D;
 3. wire W, X, Y, Z;
 4.
 5. assign Y = D[1] ∧ D[0];
 6. assign Z = X ∧ Y;
 7. assign W = ∼Z;
 8. always @(posedge clk) begin
 9.    D[1] = W;
10.    D[0] = Z;
11. end
12.
13. always #50 clk = ∼clk;
14. initial
15. begin
```

```
16.    clk = 0;
17.    D = 2'b00;
18.    #100 X = 1'b1;
19.    #100 X = 1'b0;
20.    #100 X = 1'b1;
21.    #100 X = 1'b0;
22. end
23. endmodule
```

Please first draw the corresponding finite state machine and then calculate both the state and the arc coverage from the simulation.

**9.5.** (**Equivalence Checking**) Determine whether the following two combinational circuits are functionally equivalent. If not, produce a counterexample.

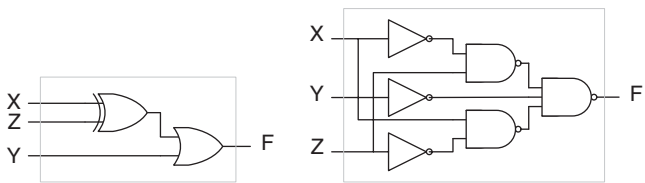



**FIGURE 9.35**

Gate-level schematics for the two circuits in Exercise 9.5.

**9.6.** (**Equivalence Checking**) Determine whether the following two sequential circuits are functionally equivalent. If not, produce a counterexample. Note that the initial states of all flip-flops are zero.

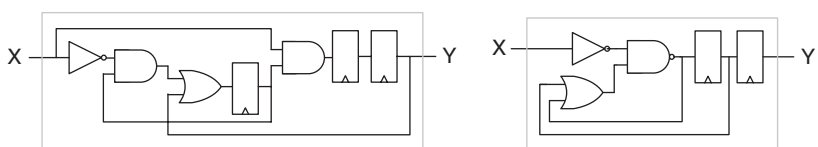



**FIGURE 9.36**

Gate-level schematics for the two circuits in Exercise 9.6.

**9.7.** (**Kripke Structure**) Derive the Kripke structure for the following circuit.

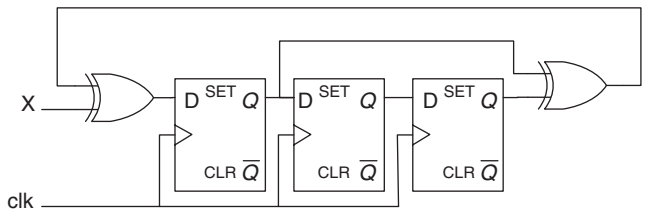



**FIGURE 9.37**

Gate-level schematic used for Exercise 9.7.

**9.8.** (**Kripke Structure**) Derive the Kripke structure for the following circuit.

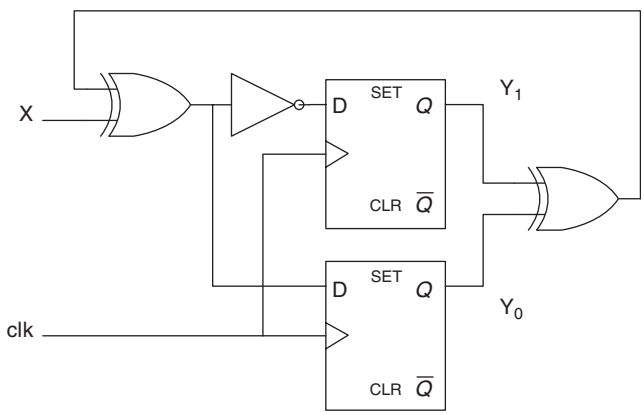



**FIGURE 9.38**

Gate-level schematic used for Exercise 9.8.

**9.9.** (**Model Checking**) Assume that $\phi$, $\psi$, and $\gamma$ are atomic propositions. Please use LTL to describe the following design properties:

**(a)** If $\psi$ occurs, $\gamma$ never occurs in the future.

**(b)** Always if $\phi$ occurs, then eventually $\psi$ occurs immediately followed by $\gamma$.

**(c)** Any occurrence of $\phi$ is followed eventually by an occurrence of $\psi$. Furthermore, $\gamma$ never occurs between $\phi$ and $\psi$.

**9.10.** **(Model Checking)** Prove or disprove the following equivalences of all LTL formulas:

**(a)** $\phi \mathbf{W} \psi \equiv \phi \mathbf{U} \psi \vee \mathbf{G} \phi$

**(b)** $\phi \mathbf{R} \psi \equiv \phi \mathbf{W} (\phi \vee \psi)$
**(c)** $\phi \mathbf{U} \psi \equiv \psi \mathbf{R} (\phi \vee \psi)$

**9.11. (Model Checking)** Prove the following equivalences of all CTL formulas:
**(a)** $\mathbf{AG}\phi \equiv \phi \wedge \mathbf{AXAG}\phi$
**(b)** $\mathbf{EF}\psi \equiv \psi \vee \mathbf{EXEF}\psi$
**(c)** $\mathbf{E}[\phi \mathbf{U} \psi] \equiv \psi \vee (\phi \wedge \mathbf{EXE}[\phi \mathbf{U} \psi])$

**9.12. (Model Checking)** Consider the model ***M*** in Figure 9.39. Please check whether $s_0 \models \phi$ and $s_3 \models \phi$ hold the following CTL formulas $\phi$'s in ***M***:
**(a)** **AG(AF***a***)**
**(b)** **EX(EX***c***)**
**(c)** **AG(EF(***c***∨***d***))**

**9.13. (Model Checking)** Assume that $\phi$ is an atomic proposition. Please prove or disprove that the formula $\mathbf{EGF}\phi$ in CTL* is equivalent to the formula $\mathbf{EGEF}\phi$ in CTL.

**9.14. (Theorem Proving)** The exclusive-or function XOR can be defined as $f = x \otimes y = \bar{x}y + x\bar{y}$ in Figure 9.40a, and its implementation is shown in Figure 9.40b. Please derive *SPEC*(*x, y, z*) from *IMPL*(*x, y, z*) by applying the inference rules specified in Table 9.3.

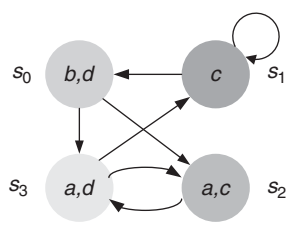



**FIGURE 9.39**

Finite state machine for the model *M* used for Exercise 9.12.

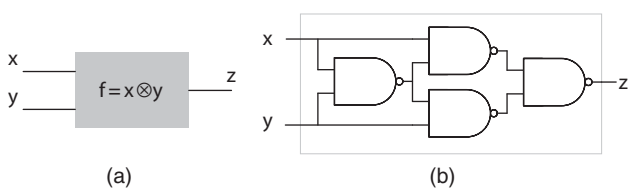



**FIGURE 9.40**

Specification and implementation views in Exercise 9.14. (a) *SPEC*(*x*,*y*,*z*). (b) *IMPL*(*x*,*y*,*z*).

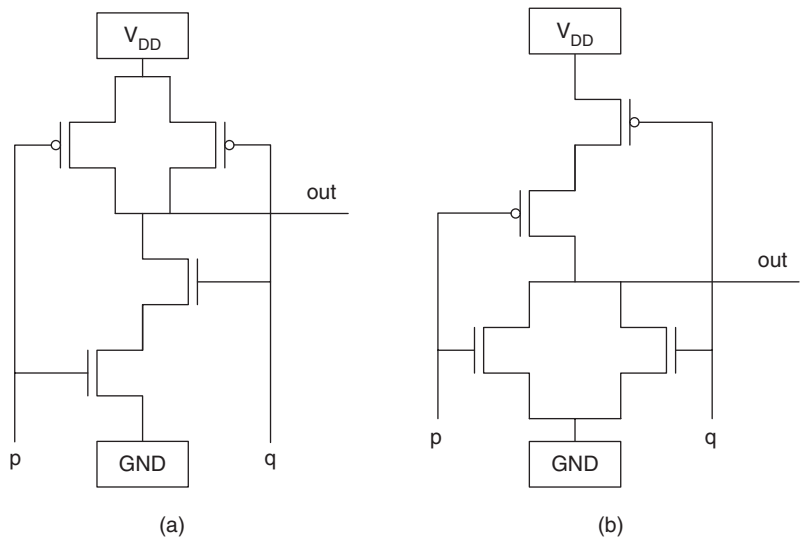



**FIGURE 9.41**

Transistor schematics for NAND and NOR gates in Exercise 9.16. (a) a NAND gate. (b) a NOR gate.

**9.15.** (**Theorem Proving**) Given $i_1$, $i_2$ as input ports and *out* as an output port, the formal specifications for NAND and XOR gates can be represented as:

- NAND($i_1$,$i_2$,*out*) $\triangleq$ $out = \neg(i_1 \wedge i_2)$
- XOR($i_1$,$i_2$,*out*) $\triangleq$ $out = (i_1 \wedge \neg i_2) \vee (\neg i_1 \wedge i_2)$

**(a)** Derive the formal descriptions for the two circuits in Exercise 9.5.

**(b)** Prove that the two circuits are equivalent by applying inference rules specified in Table 9.3.

**9.16.** (**Theorem Proving**) Given $i_1$ and $i_2$ as input ports and *out* as an output port, the formal specifications for NAND and NOR gates are:

- NAND($i_1$, $i_2$, *out*) $\triangleq$ $out = \neg(i_1 \wedge i_2)$
- NOR($i_1$, $i_2$, *out*) $\triangleq$ $out = \neg(i_1 \vee i_2)$

**(a)** Derive the formal specifications for a NAND gate from the CMOS implementation in Figure 9.41a.

**(b)** Derive the formal specifications for a NOR gate from the CMOS implementation in Figure 9.41b.

## ACKNOWLEDGMENTS

We thank Professor Michael S. Hsiao of Virginia Tech, Professor Jing-Yang Jou of National Chiao Tung University, and Professor Jie-Hong (Roland) Jiang of National Taiwan University for reviewing the text and providing helpful comments.

## REFERENCES

### R9.0 Books

[Bailey 2007] G. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology,* Morgan Kaufmann, San Francisco, February 2007.

[Bergeron 2000] J. Bergeron, *Writing Testbenches, Function Verification of HDL Models,* Second edition, Kluwer Academic Publishers, New York, February 2003.

[Dempster 2002] D. Dempster and M. Stuart, *Verification Methodology Manual: Techniques for Verifying HDL Designs,* Third Edition, Teamwork International, Hampshire, UK, June 2002.

[Foster 2004] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design,* Second Edition, Kluwer Academic Publishers, New York, May 2004.

[Gorden 1993] M. J. C. Gorden and T. F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic,* Cambridge University Press, London, June 1993.

[Granas 2003] A. Granas and J. Dugundji, *Fixed Point Theory,* Springer, New York, June 2003.

[Huth 2004] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems,* Second Edition, Cambridge University Press, New York, June 2004.

[James 2003] P. James, *Verification Plans: The Five-Day Verification Strategy for Modern Hardware Verification Languages,* Kluwer Academic Publishers, New York, October 2003.

[Nipkow 2002] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Springer-Verlag, Berlin Heidelberg, May 2002.

[Palnitkar 2003a] S. Palnitkar, *Verilog® HDL: A Guide to Digital Design and Synthesis,* Second Edition, Prentice Hall PTR, New Jersey, March 2003.

[Palnitkar 2003b] S. Palnitkar, *Design Verification with e,* Prentice Hall PTR, New Jersey, October 2003.

[Piziali 2004] A. Piziali, *Functional Verification Coverage Measurement and Analysis,* Springer, New York, October 2004.

[Prior 1957] A. N. Prior, *Time and Modality,* Clarendon Press, Oxford, 1957.

### R9.1 Introduction

[ANSI/ASQC 1978] ANSI/ASQC A3, Quality systems terminology. *American Society for Quality Control,* Milwaukee, WI, 1978.

[Bailey 2002] B. Bailey, The wake of the sleeping giant-verification, *Scalable Verification Technical Publications,* http://www.mentor.com, April 2002.

[Piziali 2006] A. Piziali, Verification planning to functional closure of processor-based SoCs, in *Proc. DesignCon*, 3-TP2, February 2006.

### R9.2 Verification Hierarchy

[Scafidi 2004] C. Scafidi, J. D. Gibson, and R. Bhatia, Validating the Itanium 2 exception control unit: A unit-level approach, *IEEE Design & Test of Computers*, 21(2), pp. 94–101, March 2004.

## R9.3 Measuring Verification Quality

[Benjamin 1999] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets, A study in coverage-driven test generation, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 970–975, June 1999.

[Drucker 2002] L. Drucker, Functional coverage metrics—the next frontier, *EETimes,* http://www.eetimes.com, August 2002.

[Gluska 2003] A. Gluska, Coverage-oriented verification of Banias, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 280–284, June 2003.

[Verisity 2001] Verisity Design Inc., *Coverage-Driven Functional Verification,* White Paper, http://www.verisity.com, 2001.

## R9.4 Simulation-Based Approach

[Accellera 2002a] Accellera, http://www.systemverilog.org, 2002

[Accellera 2002b] Accellera, http://www.accellera.org, 2002

[Synopsys 2001] Synopsys, http://www.open-vera.com, 2001

## R9.5 Formal Approaches

[Abdulla 2000] P. A. Abdulla, P. Bjesse, and N. Eén, Symbolic reachability analysis based on SAT solvers, in *Proc. 6th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 411–425, March 2000.

[Biere 1999] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, Symbolic model checking without BDDs, in *Proc. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 193–207, March 1999.

[Brand 1993] D. Brand, Verification of large synthesized designs, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Designs*, pp. 534–537, November 1993.

[Burch 1990] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, Symbolic model checking: $10^{20}$ states and beyond, in *Proc. IEEE Symp. on Logic in Computer Science*, pp. 1–33, June 1990.

[Calypto 2008] Calypto Design Systems, *SLEC System,* http://www.calypto.com, 2008.

[Clarke 1986] E. M. Clarke, E. A. Emersion, and A. P. Sistla, Automatic verification of finite state concurrent system using temporal logic specifications, *ACM Trans. on Programming Languages and System*, 8(2), pp. 144–163, April 1986.

[Cohn 1988] A. Cohn, Correctness properties of the VIPER block model: The second level, *Technical Report No. 134*, University of Cambridge, Computer Laboratory, May 1988.

[Coq 2003] The Coq Development Team, *The Coq Proof Assistant Reference Manual*, version 7.4, INRIA, http://coq.inria.fr/doc/main.html, February 2003 .

[Emerson 1990] E. A. Emerson, Temporal and modal logic, in *Handbook of Theoretical Computer Science*, Vol. B, Elsevier, pp. 996–1072, 1990.

[Goldberg 2000] E. Goldberg, M. Prasad, and R. Brayton, Using SAT for combinational equivalence checking, in *Proc. Int. Workshop on Logic Synthesis*, pp. 185–191, May 2000.

[Huang 2000] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, C.-Y. Huang, and F. Brewer, AQUILA: An Equivalence Checking System for Large Sequential Designs, *IEEE Trans. on Computers*, 49(5), pp. 443–464, May 2000.

[Joyce 1986] J. J. Joyce, G. Birtwistle, and M. Gordon, Proving a computer correct in higher order logic, *Technical Report No. 134*, University of Cambridge, Computer Laboratory, 1986.

[Kaufmann 2002] M. Kaufmann and J. Moore, A computational logic for applicative common lisp, in *A Companion to Philosophical Logic*, pp. 724–741, Blackwell Publishers, 2002.

[Kripke 1963] S. A. Kripke, Semantic consideration on modal logic, in *Proc. A Colloquium: Model and Many Valued Logic, Acta Philosophica Fennica*, 16, pp. 83–94, August 1963.

[Kunz 1993] W. Kunz, HANNIBAL: An efficient tool for logic verification based on recursive learning, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 538–543, November 1993.

[Lu 2003] F. Lu, L.-C. Wang, K.-T. Cheng, and R. C.-Y. Huang. A circuit SAT solver with signal correlation guided learning, in *Proc. IEEE/ACM Design, Automation and Test in Europe Conf.*, pp. 892–897, March 2003.

[Martin 2004] A. Martin, Adequate sets of temporal connectives in CTL, *Elsevier Electronic Notes in Theoretical Computer Science*, 52(1), pp. 1–11, January 2004.

[McMillan 1992] K. L. McMillan, *Symbolic Model Checking—An Approach to the State Explosion Problem,* PhD thesis, SCS, Carnegie Mellon University, 1992.

[Melham 1988] T. F. Melham, Abstraction mechanisms for hardware verification, in *VLSI Specification, Verification, and Synthesis*, pp. 129–157, Kluwer Academic Publishers, Boston, 1988.

[Owicki 1982] S. Owicki and L. Lamport, Proving liveness properties of concurrent programs, *ACM Trans. on Programming Languages and Systems*, 4(3), pp. 455–495, July 1982.

[Owre 1992] S. Owre, J. M. Rushby, and N. Shankar, PVS: A prototype verification system, in *Proc. 11th Int. Conf. on Automated Deduction* (*CADE*), pp. 748–752, June 1992.

[Saaltink 1999] M. Saaltink, The Z/EVES Users Guide, *Technical Report TR-97-5493-06*, ORA, Canada, 1999.

## R9.6 Advanced Research

[Bai 2003] X. Bai, L. Chen, and S. Dey, Software-based self-test for crosstalk in processors, in *Proc. Int. Workshop on High Level Design Validation and Test*, pp. 11–16, November 2003.

[Bose 2001] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir, A genetic approach to automatic bias generation for biased random instruction generation, in *Proc. 2001 Congress on Evolutionary Computation*, pp. 442–448, May 2001.

[Chen 2003] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, A scalable software-based self-test methodology for programmable processors, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 548–553, June 2003.

[Fine 2003] S. Fine and A. Ziv, Coverage directed test generation for functional verification using Bayesian networks, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 286–291, June 2003.

[Fine 2006] S. Fine, A. Freund, I. Jaeger, Y. Mansour, Y. Naveh, and A. Ziv, Harnessing machine learning to improve the success rate of stimuli generation, *IEEE Trans. on Computers*, 55(11), pp. 1344–1355, November 2006.

[Ganai 2001] M. Ganai, P. Yalagandula, A. Aziz, A. Kuehlmann, and V. Singhal, SIVA: A system for coverage-directed state space search, *J. of Electronic Testing: Theory and Applications*, 17(1), pp. 11–27, February 2001.

[Geist 1996] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal, Coverage-directed test generation using symbolic techniques, in *Proc. Int. Conf. on Formal Methods in Computer-Aided Design*, pp. 143–158, November 1996.

[Guttman 1991] J. D. Guttman, A proposed interface logic for verification environments, *Technical Report M91-19*, the MITRE Corporation, March 1991.

[Hazelhurst 2002] S. Hazelhurst, G. Kamhi, O. Weissberg, and L. Fix, A hybrid verification approach: Getting deep into the design, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 111–116, June 2002.

[Ho 1995] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill, Architecture validation for processors, in *Proc. Int. Symp. on Computer Architecture*, pp. 404–413, May 1995.

[Huang 2001] C.-Y. Huang and K.-T. Cheng, Using word-level ATPG and modular arithmetic constraint-solving techniques, *IEEE Trans. on Computer-Aided Design*, 20(3), pp. 381–391, March 2001.

[Iwashita 1994] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose, Automatic test program generation for pipelined processors, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 580–583, November 1994.

[Jang 1997] J.-Y. Jang, S. Qadeer, M. Kaufmann, and C. Pixley, Formal verification of FIRE: A case study, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 173–177, June 1997.

[Joyce 1993] J. J. Joyce and C. H. Seger, Linking BDD-based symbolic evaluation to interactive theorem-proving, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 469–474, June 1993.

[Lai 2000] W.-C. Lai, A. Krstic, and K.-T. Cheng, Functionally testable path delay faults on a microprocessor, *IEEE Design & Test of Computers*, 17(4), pp. 6–14, October 2000.

[Mishra 2005] P. Mishra and N. Dutt, Functional coverage driven test generation for validation of pipelined processors, in *Proc. IEEE/ACM Design, Automation and Test in Europe Conf.*, pp. 678–683, March 2005.

[Moondanos 1998] D. Moondanos, J. A. Abraham, and Y. V. Hoskote, Abstraction techniques for validation coverage analysis and test generation, *IEEE Trans. on Computers*, 47(1), pp. 2–14, January 1998.

[Shyam 2006] S. Shyam and V. Bertacco, Distance-guided hybrid verification with GUIDO, in *Proc. IEEE/ACM Design, Automation and Test in Europe Conf.*, pp. 1211–1216, March 2006.

[Tasiran 2001] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer, A functional validation technique: Biased-random simulation guided by observability-based coverage, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 82–88, September 2001.

[Ur 1999] S. Ur and Y. Yadin, Micro architecture coverage directed generation of test programs, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 175–180, June 1999.

[Vedula 2004] V. M. Vedula, W. J. Townhead, and J. A. Abraham, Program slicing for ATPG-based property checking, in *Proc. Int. Conf. on VLSI Design*, pp. 591–596, January 2004.

[Wagner 2005] I. Wagner, V. Bertacco, and T. Austin, StressTest: An automatic approach to test generation via activity monitors, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 783–788, June 2005.

[Wen 2006] H.-P. Wen, L.-C. Wang, and K.-T. Cheng, Simulation-based functional test generation for embedded processors, *IEEE Trans. on Computers*, 55(11), pp. 1–9, November 2006.

[Wen 2007] H.-P. Wen, L.-C. Wang, and J. Bhadra, An incremental learning framework for estimating signal controllability in unit-level verification, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 250–257, November 2007.

[Yang 1998] C. H. Yang and D. Dill, Validation with guided search of the state space, in *Proc. ACM/IEEE Design Automation Conf.*, pp. 599–604, June 1998.